A Formalisation of Nominal α-Equivalence with A, C, and AC Function Symbols ☆

Mauricio Ayala-Rincón[†,‡], Washington de Carvalho-Segundo[‡], Maribel Fernández[*], Daniele Nantes-Sobrinho[†] and Ana Cristina Rocha-Oliveira[‡1]

Departamentos de [†]Matemática e [‡]Ciência da Computação
Universidade de Brasília, Brazil

[*]*Department of Informatics, King's College London, UK*

Abstract

This paper describes a formalisation in Coq of nominal syntax extended with associative (A), commutative (C) and associative-commutative (AC) operators. This formalisation is based on a natural notion of nominal α-equivalence, avoiding the use of an auxiliary *weak* α-relation used in previous formalisations of nominal AC equivalence. A general α-relation between terms with A, C and AC function symbols is specified and formally proved to be an equivalence relation. As corollaries, one obtains the soundness of α-equivalence modulo A, C and AC operators. General α-equivalence problems with A operators are log-linearly bounded in time while if there are also C operators they can be solved in $O(n^2 \log n)$; nominal α-equivalence problems that also include AC operators can be solved with the same running time complexity as in standard first-order AC approaches.

This development is a first step towards verification of nominal matching, unification and narrowing algorithms modulo equational theories in general.

Keywords: Nominal syntax; Alpha Equivalence, Equivalence modulo A, C and AC axioms.

☆Work supported by the Brazilian agencies FAPDF (DE 193.001.369/2016), CAPES (Proc. 88881.132034/2016-01, second author) and CNPq (PQ 307672/2017-4 and universal grant 476952/2013-1), first author

[1]Email: ayala@unb.br, wtonribeiro@gmail.com, maribel.fernandez@kcl.ac.uk, dnantes@mat.unb.br, anacrismarie@gmail.com

1. Introduction

Equational problems are first-order formulas involving only one predicate: equality. Checking the validity of equational problems is a fundamental issue in automatic deduction. In this paper we focus on a particular class of equational problems: universally quantified conjunctions of equations. We aim at checking their validity modulo equational theories such as *α-equivalence, commutativity, associativity, idempotence*, etc. More generally, we consider nominal syntax instead of first-order syntax (to take into account binding operators) and assume that some function symbols obey equational axioms.

The notions of binding and α-equivalence play a fundamental role in programming languages and computation models. For example, in the *λ-calculus* [1], α-equivalence captures the notion of irrelevance of the names used as *bound variables*. At a first glance it seems to be an abstract problem but concrete examples can be provided in different syntactic computational frameworks, where a simple *renaming* of variables results in syntactically different but α-equivalent expressions. The simplest example in the λ-calculus is given by α-equivalent terms for the identity: $\lambda x.x \approx_\alpha \lambda y.y$; also, in computational languages it is enough to rename the names of the parameters of a function definition to obtain α-equivalent definitions.

Adequate manipulation of bound variables was a main motivation for the development of Nominal Logic [2] and it was taken as the basis of a series of formal developments including, *nominal unification* [3, 4, 5, 6, 7], that is, unification modulo $\approx_\alpha$, *nominal rewriting* [8, 9, 10], *deduction systems* [11], *programming languages* [12, 13, 14] and *reasoning frameworks* [15, 16].

In nominal syntax, instead of variables one uses *atoms* that are distinguished by their names and used to build abstractions. Additionally, the notion of *freshness* is made explicit through inference rules that define whether atoms are *free* or not in a nominal term. *Renaming* of variables is defined through *swappings* of atoms that are essential components of *permutations acting* over terms. Finally, the notion of α-equivalence is axiomatised through inference rules that specify whether, under some freshness constraints, terms are α-equivalent or not. This differs from the usual treatment in frameworks such as the λ-calculus, where α-equivalence is implicitly abstracted through assumptions such as Barendregt's variable convention [1].

The best known and most complete formal development of nominal syntax

was specified in Isabelle/HOL by Urban et al. ([6, 7]): firstly, a relation $\approx_\alpha$ is specified and proved to be sound, that is, proved to be an equivalence relation; secondly, a nominal unification algorithm is specified, which uses α-equivalence, and verified to be correct and complete. In particular, Urban [6] describes in detail how to prove that the nominal $\approx_\alpha$ relation is in fact an equivalence relation using an intermediate *weak* α-relation denoted as $\sim_\omega$. This technique was introduced by Kumar and Norrish [4] in a HOL4 formalisation of nominal unification, and was also applied in a previous version of our formalisation [17]. In this paper, we present an even simpler proof, avoiding formalisations of properties of this weak intermediate relation. This is obtained following the analytic scheme of proof shown in [8] and first applied in the PVS formalisation of nominal unification in [5].

Contribution. This paper describes a formalisation in the Coq proof assistant of the soundness of α-equivalence in nominal syntax. The distinguishing feature of this development is that we advance further and also check nominal α-equivalence with combinations of A, C and AC operators. The development can be extended to other equational theories. The main steps of the formalisation are described below.

- Initially, the notion of α-equivalence $\approx_\alpha$ is specified and proved to be sound. Although this property is usually taken for granted, its formalisation is not straightforward, since it relies on a non trivial induction on terms in which the induction hypothesis cannot be directly established for *convenient (α) renaming of proper sub-terms of the term to which the induction is applied.* Other crucial, but non-trivial properties are necessary: *preservation of freshness*, *equivariance* of $\approx_\alpha$, preservation of the action of *permutations*, etc.

- Then, α-equivalence with A, C and AC operators, denoted as $\approx_{\{A,C,AC\}}$, is specified and proved sound. The soundness of α-equivalence modulo A ($\approx_{\alpha,A}$), C ($\approx_{\alpha,C}$) and modulo AC ($\approx_{\alpha,AC}$) are inferred from the soundness of $\approx_{\{A,C,AC\}}$. These relations are specified in a parameterised manner, which will simplify the treatment and combination of α-equivalence with other equational theories. More precisely, the set of countable function symbols used to build terms in nominal syntax is annotated using *scripts*: A *superscript* distinguishes the equational properties of the operator, and a *subscript* gives the index of the function symbol in the class of symbols with the same equational properties; thus, for instance, f_k^{AC} denotes the k^{th} AC function symbol in the signature. The relation

$\approx_{\{A,C,AC\}}$ is defined using the rules of α-equivalence and it is proved that its restriction to α-equivalence yields $\approx_\alpha$. Thus, using correctness of $\approx_\alpha$, the relation $\approx_{\{A,C,AC\}}$ is checked by applying the algebraic properties of A, C and AC operators and, in addition properties of *preservation of freshness* and *equivariance* for $\approx_{\{A,C,AC\}}$.

A naive OCaml implementation of the decision algorithm for $\approx_{\{A,C,AC\}}$ has been automatically extracted from the Coq specification. Also, an improved version is given that deals more efficiently with arguments of associative operators by flattening them, and avoids unnecessary comparisons between arguments of AC operators when checking equality. Experiments were performed comparing the extracted and the improved implementations over randomly generated equational problems. When checking equivalence, the decision whether one should or should not apply nominal inference rules specialised for A, C or AC symbols is done in a natural manner using the superscript of the function symbols.

Regarding complexity, assuming a pre-computation of the flat form of terms headed with A and AC function symbols, and efficient data structures for manipulation of nominal terms and permutations, such as those used for the implementation of nominal α-equivalence and matching in [18], the following results are proved:

- Deciding α-equivalence modulo A only is log-linear in time on the size of the problem (i.e., $O(n \log n)$);
- If there are only A operators and C operators, then the complexity is $O(n^2 \log n)$; and
- α-equivalence modulo (A, C and) AC can be decided by adapting the matching algorithm presented by Benanav, Kapur and Narendran [19] for the case of pure AC-equivalence in standard first-order syntax, obtaining an $O(n^3 \log n)$ upper bound.

Outline. Section 2 presents necessary background on nominal abstract syntax. Sections 3 and 4 respectively present the formalisations of soundness of α-equivalence and its version with A, C and AC operators. Section 5 discusses experiments with the OCaml extracted and improved implementations, and gives complexity bounds for the problem of deciding $\approx_{\{A,C,AC\}}$. Before concluding, Section 6 presents related work. The Coq specification is available at `http://ayala.mat.unb.br/publications.html`.

2. Nominal Syntax

This section presents necessary notions and notations of nominal syntax [8].

Given a signature Σ of function symbols and $\mathcal{V}$ and $\mathcal{A}$ countably infinite sets of *variables* and *atoms*, the set $\mathcal{T}(\Sigma, \mathcal{A}, \mathcal{V})$ of *nominal terms* is generated by the following grammar:

$$s, t \ ::= \ \langle\rangle \ \mid \ \overline{a} \ \mid \ [a]t \ \mid \ \langle s, t\rangle \ \mid \ f_k^E\, t \ \mid \ \pi.X$$

Atoms are the simplest structure, just object-level variables $a \in \mathcal{A}$. Atoms only differ in their names, so for atoms a and b the expression $a \neq b$ is redundant. A *permutation* is a bijection on $\mathcal{A}$ with a finite domain. A *swapping* is defined as a pair of atoms $(a\, b)$ and a *permutation* π is represented by a finite list of *swappings* of the form $(a_1\, b_1) :: \ldots :: (a_n\, b_n) :: \mathit{nil}$, where *nil* denotes the identity permutation. The composition of permutations π and π' is denoted as $\pi' \oplus \pi$. Unary permutations $(a\, b) :: \mathit{nil}$ will be abbreviated as $(a\, b)$. A variable $X \in \mathcal{V}$ as a term object should always be decorated by some permutation π *suspended* on X, $\pi.X$. For brevity, terms of the form $\mathit{nil}.X$ will be written as X.

Definition 1. *The* size *of a term t, denoted as $|t|$, is recursively defined as:*

$$|[a]t| \ := \ |t| + 1, \ \ |\langle u, v\rangle| \ := \ |u| + |v| + 1, \ \ |f_k^E\, s| \ := \ |s| + 1, \ \ \_ \ := \ 1.$$

Permutations *act* on nominal terms, but suspend over variables. The *empty tuple* or *unit* is denoted as $\langle\rangle$ and non empty tuples are built using *pairs* of terms of the form $\langle s, t\rangle$, where s and t might be also pairs. Notice that this syntax does not allow construction of unary tuples. The notation $\overline{a}$ represents the atom a as a term object. $[a]t$ is an *abstraction* of an atom a in a term t. The notation $f_k^E\, t$ represents the *application* of $f_k^E \in \Sigma$ to t. The scripts E and k in the function symbol f_k^E are respectively used to distinguish the equational properties of the function symbol and the indexation of the function symbol between the class of operators with the same equational properties. These scripts will be omitted when no confusion arises.

In the Coq specification the grammar is written as in Figure 1. Operators `Ut, At, Ab, Pr, Fc` and `Su` specify the unit, atoms as term objects, abstractions, pairs, function applications and suspended variables, respectively. For the `Fc` constructor, the first and second `nat` arguments represent the super and subscripts of the applied function symbol. In the formalisation, the function symbols f_j^A, f_k^{AC} and f_n^C are represented respectively by `Fc 0 j`,

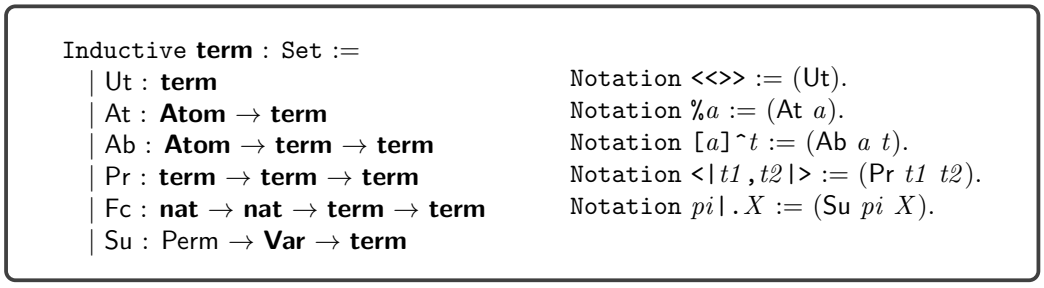

Figure 1: Coq specification of the grammar of terms

`Fc 1 k` and `Fc 2 n`, all having type `term` → `term`. All other superscripts are representing the empty equational theory.

Although in nominal syntax different atoms a and b are assumed to be different, this is not automatically true in computational specifications. Indeed, since the given approach uses metavariables ranging over naturals to represent atoms, different variables might represent the same atom. Then, rules ($\#\,\mathbf{a}[\mathbf{b}]$) and ($\approx_\alpha\,[\mathbf{ab}]$), respectively, from Figures 2 and 3, were specified with the extra condition $a \neq b$.

Definition 2. *The* action *of a permutation over terms is specified as the homeomorphic extension of the action of lists of swappings over single atoms:*

$$\begin{array}{llll} \pi \cdot \langle\rangle := \langle\rangle & \pi \cdot \langle u, v\rangle := \langle \pi \cdot u, \pi \cdot v\rangle & \pi \cdot f_k^E\, t := f_k^E\,(\pi \cdot t) \\ \pi \cdot \overline{a} := \overline{\pi \cdot a} & \pi \cdot ([a]t) := [\pi \cdot a](\pi \cdot t) & \pi \cdot (\pi'\,.\,X) := (\pi' \oplus \pi)\,.\,X \end{array}$$

The action of a permutation over an atomic term object $\overline{a}$, e.g., $\pi \cdot \overline{a}$, gives as result a term $\overline{\pi \cdot a}$. This is specified as $\pi \cdot (\texttt{At}\ a)$, which gives as result $\texttt{At}\ (\pi \cdot a)$, and not the atom $\pi \cdot a$.

The action of the permutation π over the suspended variable $\pi'.X$ gives as result the term $\pi \cdot (\pi'.X) = (\pi' \oplus \pi).X$. Notice that permutation composition works in the opposite direction.

Example 1. *The permutation* $(a\ b) :: \pi$ *acting over the term* $[a]\langle \overline{b}, \pi'.X\rangle$ *will have as result* $[\pi \cdot b]\langle \overline{\pi \cdot a}, (\pi' \oplus ((a\ b) :: \pi)).X\rangle$.

2.1. Freshness and $\boldsymbol{\alpha}$*-equivalence*

The native notion of equality on nominal terms is α-equivalence, which is defined using *swappings* and a notion of freshness. A *freshness constraint* is a pair $a \# t$ of an atom and a nominal term t. Intuitively, $a \# t$ means that a is

fresh in t, that is, if a occurs in t then it must do so under an abstractor $[a]$. An *α-equality constraint* is a pair $s \approx_\alpha t$ of two terms s and t. A *freshness context*, is a set of *freshness constraints* whose elements are restricted to pairs $a\#X \in \mathcal{A} \times \mathcal{V}$. ∇ will range over freshness contexts. A *freshness judgement* is a tuple of the form $\nabla \vdash a \# t$, whereas an *α-equivalence judgement* is a tuple of the form $\nabla \vdash s \approx_\alpha t$.

$$\frac{}{\nabla \vdash a \# \langle\rangle}\,(\#\langle\rangle) \quad \frac{}{\nabla \vdash a \# \bar{b}}\,(\#\,\mathbf{atom}) \quad \frac{\nabla \vdash a \# t}{\nabla \vdash a \# f_k^E\, t}\,(\#\,\mathbf{app}) \quad \frac{}{\nabla \vdash a \# [a]t}\,(\#\,\mathbf{a[a]})$$

$$\frac{\nabla \vdash a \# t}{\nabla \vdash a \# [b]t}\,(\#\,\mathbf{a[b]}) \quad \frac{(\pi^{-1} \cdot a\#X) \in \nabla}{\nabla \vdash a \# \pi.X}\,(\#\,\mathbf{var}) \quad \frac{\nabla \vdash a \# s \;\; \nabla \vdash a \# t}{\nabla \vdash a \# \langle s, t\rangle}\,(\#\,\mathbf{pair})$$

Figure 2: Rules for the freshness relation

The *derivable* freshness and α-equivalence judgements are defined by the rules in Figures 2 and 3 (cf. Figure 2 in [7]). We write $ds(\pi, \pi')\#X$ as an abbreviation of $\{a \# X \mid a \in ds(\pi, \pi')\}$, where $ds(\pi, \pi') = \{a \mid \pi \cdot a \neq \pi' \cdot a\}$ is the set of atoms where π and π' differ (the *difference set*). A set $\mathcal{P}$ of constraints is called a *problem*. We write $\nabla \vdash \mathcal{P}$ when proofs of the judgment $\nabla \vdash P$ exist for each $P \in \mathcal{P}$, using rules of Figures 2 and 3.

$$\frac{}{\nabla \vdash \langle\rangle \approx_\alpha \langle\rangle}\,(\approx_\alpha \langle\rangle) \quad \frac{}{\nabla \vdash \overline{a} \approx_\alpha \overline{a}}\,(\approx_\alpha \mathbf{atom}) \quad \frac{\nabla \vdash s \approx_\alpha t}{\nabla \vdash f_k^E\, s \approx_\alpha f_k^E\, t}\,(\approx_\alpha \mathbf{app})$$

$$\frac{\nabla \vdash s \approx_\alpha t}{\nabla \vdash [a]s \approx_\alpha [a]t}\,(\approx_\alpha [\mathbf{aa}]) \quad \frac{\nabla \vdash s \approx_\alpha (a\,b) \cdot t \;\; \nabla \vdash a \# t}{\nabla \vdash [a]s \approx_\alpha [b]t}\,(\approx_\alpha [\mathbf{ab}])$$

$$\frac{ds(\pi, \pi')\#X \subseteq \nabla}{\nabla \vdash \pi.X \approx_\alpha \pi'.X}\,(\approx_\alpha \mathbf{var}) \quad \frac{\nabla \vdash s_0 \approx_\alpha t_0 \;\; \nabla \vdash s_1 \approx_\alpha t_1}{\nabla \vdash \langle s_0, s_1\rangle \approx_\alpha \langle t_0, t_1\rangle}\,(\approx_\alpha \mathbf{pair})$$

Figure 3: Rules for the relation $\approx_\alpha$

The interesting rules for freshness are those for abstractions and suspensions. For example, $\nabla \vdash a \# \langle [a](\langle \overline{a}, \overline{b}\rangle), \pi.X\rangle$ can be derived only if the pair $\pi^{-1} \cdot a\#X$ is in the context ∇, where π^{-1} is the reverse list of π.

The interesting inference rules for α-equivalence are those for abstractions and suspended variables. For abstraction we have two possible cases: $(\approx_\alpha [\mathbf{aa}])$ and $(\approx_\alpha [\mathbf{ab}])$. In the former case, one needs to check whether the abstracted terms are α-equivalent under the same context, and in the latter case, when the abstraction is built with different atoms, one needs to check whether

renaming one of the abstracted terms by swapping these different atoms, the α-equivalence with the other abstracted term holds, in addition, the new atom has to be fresh in the abstracted term that is renamed. From the nominal syntax specified in Coq, the proof that `alpha_equiv` (that is, $\approx_\alpha$ of Figure 3) is in fact an equivalence relation was formalised.

3. Formalisation of soundness of the $\approx_\alpha$ relation

This section shortly describes the proofs formalised in Coq about the fact that the relation $\approx_\alpha$ given in Figure 3 is indeed an equivalence relation.

The Coq formalisation of transitivity of the relation $\approx_\alpha$ (Lemma 8) adopts a direct method, introduced in a previous PVS formalisation of nominal α-unification (see [5]). This approach avoids the use of an auxiliary weak equivalence $\sim_\omega$, introduced in the HOL4 formalisation [4] and adopted in the Isabelle/HOL formalisation [6] and in the original method formalised in Coq [17]. The more direct approach used in the current paper gives the following specific benefits: a shorter formalisation, since a series of auxiliary lemmas on $\sim_\omega$ are no longer necessary; also, intermediate transitivity lemmas relating $\sim_\omega$ and $\approx_\alpha$ are no longer necessary; the direct approach requires only a few new auxiliary lemmas on $\approx_\alpha$ that are proved by simple induction on terms and the inductive definition of the α-equality inference rules. It is also important to stress that despite the fact that the current formalisation of the lemma of transitivity of $\approx_\alpha$ is now based only on nominal properties and basic properties of $\approx_\alpha$, the proof of this lemma is not more complex than the previous one (the number of proof lines is almost the same).

Lemma 1 (Equivariance of Freshness)**.** $\nabla \vdash a \# s$ *iff* $\nabla \vdash \pi \cdot a \# \pi \cdot s$.

Lemma 2 (Freshness preservation under $\approx_\alpha$)**.** $\nabla \vdash a \# s$ *and* $\nabla \vdash s \approx_\alpha t$ *imply* $\nabla \vdash a \# t$.

Lemma 3 (Inversion of permutations over $\approx_\alpha$)**.** $\nabla \vdash \pi \cdot s \approx_\alpha t$ *implies* $\nabla \vdash s \approx_\alpha \pi^{-1} \cdot t$

Lemma 4 (Equivariance of $\approx_\alpha$)**.** $\nabla \vdash s \approx_\alpha t$ *iff* $\nabla \vdash \pi \cdot s \approx_\alpha \pi \cdot t$.

Lemma 5 (Invariance of $\approx_\alpha$ under the action of permutations)**.** $(\forall a \in ds(\pi, \pi'),\ \nabla \vdash a \# s)$ *iff* $\nabla \vdash \pi \cdot s \approx_\alpha \pi' \cdot s$.

Lemmas 1 and 3 to 5 are proved by induction on the structure of s. Lemma 2 is proved by induction on the derivation cases of $\approx_\alpha$. For Lemma 3, Lemma 2 is also applied.

Lemma 6 (Reflexivity of $\approx_\alpha$)**.** $\nabla \vdash t \approx_\alpha t$

Reflexivity of $\approx_\alpha$ is proved by induction on the structure of t.

The current proof of symmetry of $\approx_\alpha$ (Lemma 7) is independent of transitivity, unlike the previous approach using $\sim_\omega$, in which the formalisation of symmetry relies on transitivity.

Lemma 7 (Symmetry of $\approx_\alpha$)**.** *If* $\nabla \vdash s \approx_\alpha t$ *then* $\nabla \vdash t \approx_\alpha s$.

Symmetry of $\approx_\alpha$ is proved by rule induction over $\nabla \vdash s \approx_\alpha t$. The interesting case is given by rule ($\approx_\alpha$ [**ab**]). In this case, $\nabla \vdash [a]u \approx_\alpha [b]v$ whenever $\nabla \vdash u \approx_\alpha (a\,b) \cdot v$ and $\nabla \vdash a \# v$. By equivariance of freshness (Lemma 1), we obtain $\nabla \vdash b \# (a\,b) \cdot v$. By induction hypothesis (for short, IH), $\nabla \vdash (a\,b) \cdot v \approx_\alpha u$ and then $\nabla \vdash b \# u$, by Lemma 2. Finally, by inversion of permutations over $\approx_\alpha$ (Lemma 3), $\nabla \vdash v \approx_\alpha (a\,b) \cdot u$. This and $\nabla \vdash b \# u$ prove $\nabla \vdash [b]v \approx_\alpha [a]u$.

Lemma 8 (Transitivity of $\approx_\alpha$)**.** *The relation* $\approx_\alpha$ *is transitive under a given context* ∇*, i.e.,* $\nabla \vdash t_1 \approx_\alpha t_2$ *and* $\nabla \vdash t_2 \approx_\alpha t_3$ *imply* $\nabla \vdash t_1 \approx_\alpha t_3$.

Proof. The proof is by induction on the size of t_1 and case analysis over $\nabla \vdash t_1 \approx_\alpha t_2$ and $\nabla \vdash t_2 \approx_\alpha t_3$. The subsequent steps show the abstraction case, which is the most interesting one due to the asymmetry of rule ($\approx_\alpha$ [**ab**]) (see Figure 3). Consider $t_1 = [a]u$, $t_2 = [b]v$ and $t_3 = [c]w$. So one must analyse the following situations:

- $a = b = c$: thus the result follows by IH;
- $a = b \neq c$: by definition, $\nabla \vdash u \approx_\alpha v$ and $\nabla \vdash v \approx_\alpha (b\,c) \cdot w$ and $\nabla \vdash b \# w$. By IH, $\nabla \vdash u \approx_\alpha (b\,c) \cdot w$. As $a = b$, then freshness condition to a is satisfied as well;
- $a \neq b = c$: we have that $\nabla \vdash a \# v$, $\nabla \vdash u \approx_\alpha (a\,c) \cdot v$ and $\nabla \vdash v \approx_\alpha w$. By Lemma 4, $\nabla \vdash (a\,c) \cdot v \approx_\alpha (a\,c) \cdot w$ and, by IH, $\nabla \vdash u \approx_\alpha (a\,c) \cdot w$. By Lemma 1, $\nabla \vdash c \# (a\,c) \cdot v$ and $\nabla \vdash c \# (a\,c) \cdot w$ by Lemma 2. Finally, again by Lemma 1, $\nabla \vdash a \# w$;
- $b \neq a = c$: it is known that $\nabla \vdash u \approx_\alpha (b\,c) \cdot v$ and $\nabla \vdash v \approx_\alpha (b\,c) \cdot w$. Then $\nabla \vdash (b\,c) \cdot v \approx_\alpha w$ by Lemma 3. By IH $\nabla \vdash u \approx_\alpha w$;

- $a \neq b \neq c \neq a$: it is necessary to prove that $\nabla \vdash u \approx_\alpha (a\,c) \cdot w$ and $\nabla \vdash a \# w$. Let us prove first the freshness condition. by definition of $\approx_\alpha$, $\nabla \vdash a \# v$ and $\nabla \vdash v \approx_\alpha (b\,c) \cdot w$. By Lemma 2, $\nabla \vdash a \# (b\,c) \cdot w$ and, by Lemma 1, $\nabla \vdash a \# w$. Now let us prove $\approx_\alpha$: By Lemma 4, $\nabla \vdash (a\,b) \cdot v \approx_\alpha [(b\,c), (a\,b)] \cdot w$. As $ds([(b\,c), (a\,b)], (a\,c)) = \{a, b\}$ and both atoms are fresh in w, then $\nabla \vdash [(b\,c), (a\,b)] \cdot w \approx_\alpha (a\,c) \cdot w$ by Lemma 5. Now, applying IH twice, one obtains $\nabla \vdash u \approx_\alpha (a\,c) \cdot w$.

□

This approach that does not use weak equivalence, reduces considerably the effort necessary to formalise the transitivity of $\approx_\alpha$. The new strategy results in a reduction of 161 proof lines in the whole formalisation as discussed below. A few auxiliary lemmas about properties of difference sets and the relations $\#$ and $\approx_\alpha$ are necessary; among them, some lemmas that were easily proved in the former formalisation using $\approx_\alpha$-transitivity such as inversion of permutations over $\approx_\alpha$ (Lemma 3). Notice that this lemma is now necessary for proving symmetry and transitivity of $\approx_\alpha$ (see Lemmas 7, 8). Other new auxiliary lemmas specify simple properties that are now used in the inductive analysis in the proofs of symmetry and transitivity of $\approx_\alpha$, such as:

$ds(\pi, \pi') = \emptyset$ implies

1. $\nabla \vdash \pi \cdot s \approx_\alpha t$ iff $\nabla \vdash \pi' \cdot s \approx_\alpha t$;
2. $\nabla \vdash s \approx_\alpha \pi \cdot t$ iff $\nabla \vdash s \approx_\alpha \pi' \cdot t$; and
3. $\nabla \vdash a \# \pi \cdot s$ iff $\nabla \vdash a \# \pi' \cdot s$.

These lemmas are proved by induction on terms. The same technique is used in the formalisation of Lemma 3. All these results added only 177 proof lines.

On the other hand, all definitions and results about $\sim_\omega$ are no longer needed in the new approach. Statements similar to Lemmas 2 and 4 (freshness preservation and equivariance, respectively) that were proved for the weak equivalence $\sim_\omega$ are no longer necessary. Also, two auxiliary lemmas that were crucial in the former approach for the proof of transitivity of $\approx_\alpha$ were eliminated, namely: (i) $\nabla \vdash t_1 \approx_\alpha t_2$ and $t_2 \sim_\omega t_3$ implies $\nabla \vdash t_1 \approx_\alpha t_3$; and (ii) $\nabla \vdash t_1 \approx_\alpha t_2$ and $\nabla \vdash t_2 \approx_\alpha \pi \cdot t_2$ implies $\nabla \vdash t_1 \approx_\alpha \pi \cdot t_2$. Auxiliary lemma (i) establishes an intermediate transitivity combining $\approx_\alpha$ and $\sim_\omega$, and is used in the proof of auxiliary lemma (ii), as well as in the proof of equivariance, transitivity and symmetry; in all cases, it was applied for the analysis of the case of application of the rule ($\approx_\alpha$ [**ab**]). Auxiliary lemma

(ii) had a non trivial formalisation which required as much effort as the former proof of transitivity for $\approx_\alpha$. This lemma was used only in the proof of transitivity, specifically for the case of application of the rule ($\approx_\alpha$ [**ab**]). Both these auxiliary lemmas were proved by induction on the derivation rules of $\nabla \vdash t_1 \approx_\alpha t_2$. Counting all these lemmas, a total of 338 proof lines were eliminated.

Despite the facts that in the current approach the proof of symmetry (Lemma 7) is independent of the proof of transitivity (Lemma 8) and that the whole formalisation is shorter than the one using $\sim_\omega$, as previously mentioned, it is important to stress that in both approaches, the number of proof lines specifically required in the formalisations of the lemmas of symmetry and transitivity are almost the same. Indeed, in both approaches the proofs of symmetry are done by induction on the derivation rules, while the proofs of transitivity by induction on the size of terms.

Finally, to check α-equivalence modulo A, C and AC, denoted $\approx_{\{A,C,AC\}}$, one uses soundness of $\approx_\alpha$, which allows for the use of any of these approaches for checking $\approx_\alpha$, to be extended to check $\approx_{\{A,C,AC\}}$. This flexibility is obtained via the specification of an inductive relation **equiv**(S) parameterised by a set S of indices, where each index is associated to a different equational theory. In particular, the relation **equiv**$(\emptyset)$ excludes from the specification of **equiv** all specialised inference rules for any equational theory. The relation **equiv**$(\emptyset)$ is formally proved to be equivalent to the relation $\approx_\alpha$: $\nabla \vdash t \approx_\alpha t' \Leftrightarrow$ **equiv**$(\emptyset)(\nabla, t, t')$.

4. Formalising soundness of $\approx_{\{A,C,AC\}}$, $\approx_{\alpha,A}$, $\approx_{\alpha,C}$ and $\approx_{\alpha,AC}$

The generic relation **equiv** mentioned at the end of Sec. 3, will consider A, AC and C function symbols if $0, 1$ or $2 \in S$, respectively. Namely, **equiv**$(\{0\})$, **equiv**$(\{1\})$, **equiv**$(\{2\})$) and **equiv**$(\{0, 1, 2\})$ select the specialised inductive rules in the definition of **equiv** for the relation $\approx_\alpha$ modulo A, AC, C and combinations of A, AC and C, respectively. In this way one builds the relations $\approx_{\alpha,A}$, $\approx_{\alpha,AC}$, $\approx_{\alpha,C}$ and $\approx_{\{A,C,AC\}}$. For readability, from now on, instead of indices 0, 1 and 2, the corresponding abbreviations A, AC and C will be used.

4.1. Operations over tuples

The inductive rules for A and AC operators in the definition of the relation $\approx_{\{A,C,AC\}}$ use three auxiliary operators that deal with arguments of function symbols. Arguments of a function symbol f are terms or tuples built using the constructor for pairs and the arguments of terms headed by the same function symbol f. These operators, specified as in Fig. 4, extract the relevant information of the arguments

to which a(n A or AC) symbol f_n^E is applied and specify the *length or number of arguments*, $\|t\|_{f_n^E} := \mathsf{TPlength}\, t\, E\, n$, and the *selection* and *deletion of the* i^{th} *argument*, respectively, $t_{(i)_{f_n^E}} := \mathsf{TPith}\, i\, t\, E\, n$ and $t_{[\star i]_{f_n^E}} := \mathsf{TPithdel}\, i\, t\, E\, n$.

```
Fixpoint TPlength (t: term) (E n: nat) : nat :=
match t with
 | (<|t1,t2|>) ⇒ (TPlength t1 E n)
                 + (TPlength t2 E n)
 | (Fc E0 n0 t0) ⇒ if (E,n) = (E0,n0)
                   then (TPlength t0 E n)
                   else 1
 | _ ⇒ 1
end.
```

```
Fixpoint TPith (i: nat) (t: term) (E n: nat) :
term :=
match t with
 | (<|t1,t2|>) ⇒ let l1 := TPlength t1 E n in
                 if i ≤ l1
                 then TPith i t1 E n
                 else TPith (i-l1) t2 E n

 | (Fc E0 n0 t0) ⇒ if (E,n) = (E0,n0)
                   then TPith i t0 E n
                   else t

 | _ ⇒ t
end.
```

```
Fixpoint TPithdel (i: nat) (t: term) (E n: nat) : term :=
match t with
 | (<|t1,t2|>) ⇒ let l1 := (TPlength t1 E n) in
                 let l2 := (TPlength t2 E n) in
                   if i ≤ l1
                   then
                     if l1 = 1
                     then t2
                     else <|(TPithdel i t1 E n),t2|>
                   else
                     let ii := i-l1 in
                       if l2 = 1
                       then t1
                       else <|t1,(TPithdel ii t2 E n)|>

 | (Fc E0 n0 t0) ⇒ if (TPlength (Fc E0 n0 t0) E n) = 1
                   then <<>>
                   else Fc E0 n0 (TPithdel i t0 E n)

 | _ ⇒ <<>>
end.
```

Figure 4: Specification of operators for the length of the tuple or arguments, selection and deletion of the i^{th} argument regarding the function symbol f

To simplify notation, the scripts of f will be omitted in these operators when clear from the context. The behaviour of these operators is illustrated below.

Example 2. *For the number of arguments.*

1. $\|f\langle\,\rangle\|_f = \|\langle\,\rangle\|_f = 1$;

2. $\|f\,\langle \overline{a}, \overline{b}\rangle\|_f = \|\langle \overline{a}, \overline{b}\rangle\|_f = 2$, *but* $\|g\,\langle \overline{a}, \overline{b}\rangle\|_f = 1$;
3. $\|f\,\langle [a](\pi \cdot X), f\,\langle \overline{b}, g\,\langle \overline{a}, f\,\langle \overline{a}, \overline{b}\rangle\rangle\rangle\rangle\|_f =$
$\|[a](\pi \cdot X)\|_f + \|\overline{b}\|_f + \|g\,\langle \overline{a}, f\,\langle \overline{a}, \overline{b}\rangle\rangle\|_f = 3$.

Example 3. *For the selection of the* i^{th} *argument.*

1. $t_{(0)_f} = t_{(1)_f}$ *and, if* $i > \|t\|_f$ *then* $t_{(i)_f} = t_{(\|t\|_f)_f}$;
2. *If* $\|t\|_f = 1$ *and* t *is not headed by* f *then* $t_{(1)_f} = t$, *but also* $(f\,f\,t)_{(1)_f} = t$;
3. $(f\,\langle [a](\pi \cdot X), f\,\langle \overline{b}, g\,\langle \overline{a}, f\,\langle \overline{a}, \overline{b}\rangle\rangle\rangle\rangle)_{(3)_f} = (f\,\langle \overline{b}, g\,\langle \overline{a}, f\,\langle \overline{a}, \overline{b}\rangle\rangle\rangle)_{(2)_f} =$
$(g\,\langle \overline{a}, f\,\langle \overline{a}, \overline{b}\rangle\rangle)_{(1)_f} = g\,\langle \overline{a}, f\,\langle \overline{a}, \overline{b}\rangle\rangle$.

Example 4. *For the deletion of the* i^{th} *argument.*

1. $t_{[\star 0]_f} = t_{[\star 1]_f}$ *and if* $i > \|t\|_f$ *then* $t_{[\star i]_f} = t_{[\star \|t\|_f]_f}$;
2. *If* $\|t\|_f = 1$ *then* $t_{[\star 1]_f} = \langle\,\rangle$;
3. $(f\,\langle [a](\pi \cdot X), f\,\langle \overline{b}, g\,\langle \overline{a}, f\,\langle \overline{a}, \overline{b}\rangle\rangle\rangle\rangle)_{[\star 2]_f} =$
$f\,\langle [a](\pi \cdot X), (f\,\langle \overline{b}, g\,\langle \overline{a}, f\,\langle \overline{a}, \overline{b}\rangle\rangle\rangle)_{[\star 1]_f}\rangle = f\,\langle [a](\pi \cdot X), f\,(g\,\langle \overline{a}, f\,\langle \overline{a}, \overline{b}\rangle\rangle)\rangle$.

It should be clear to the reader that in the adopted syntax function symbols have no fixed arity. Thus, the application of an A or an AC function symbol to the unit ($\langle\rangle$) may be interpreted as the neutral element in the given signature. For instance, $\wedge\langle\rangle$, $\vee\langle\rangle$, $+\langle\rangle$ and $\times\langle\rangle$ might be specified as "false", "true", 0 and 1, respectively.

The use of operators $\|\_\|_f$, $\_(\_)_f$ and $\_[\star\_]_f$ has two advantages: first, neither an additional data structure to express associativity is necessary (e.g. lists, sequences, arrays) nor an operator for *flattening* terms; second, the adopted grammar permits the manipulation of arbitrary combinations of different function symbols with different equational properties, occurring simultaneously in a term, via the use of specialised rules which fit the given signature and its corresponding equational theory. This simplifies the treatment of α-equivalence modulo A, C and AC, and other equational theories.

In Table 1 a few formalised results are listed, from a much longer list of formalised lemmas related with these operators. These results will be referenced in the description of the lemmas related with E-equivalence and for brevity they are presented free of universal quantifiers.

4.2. Extension of the $\approx_\alpha$*-rules*

New rules ($\approx_\alpha$ **A**), ($\approx_\alpha$ **C**), and ($\approx_\alpha$ **AC**) for associativity, commutativity and associativity-commutativity are introduced. These rules will be combined with those from Fig. 3 for $\approx_\alpha$, with the following modification: ($\approx_\alpha$ **app**) will be replaced by ($\approx_\alpha$ $\overline{\textbf{app}}$) and applies whenever the function symbol f_k^E applied to s is such

Table 1: Basic properties of the operators over terms: $\|\_\|_f$, $\_(\_)_f$ and $\_[\star\_]_f$

$\|t\| \geq 1,\ t_{(0)} = t_{(1)},\ t_{[\star 0]} = t_{[\star 1]}$	$i \geq \|t\| \Rightarrow t_{(i)} = t_{(\|t\|)},\ t_{[\star i]} = t_{[\star \|t\|]}$
$\|t\| = 1 \Rightarrow t_{[\star i]} = \langle\rangle$	$\|t\| \neq 1 \Rightarrow \|t_{[\star i]}\| = \|t\| - 1$
$0 < i < j,\ i < \|t\| \Rightarrow (t_{[\star j]})_{(i)} = t_{(i)}$	$0 < i < j \leq \|t\| \Rightarrow (t_{[\star j]})_{[\star i]} = (t_{[\star i]})_{[\star(j-1)]}$
$0 < i < \|t\|,\ i \geq j \Rightarrow (t_{[\star j]})_{(i)} = t_{(i+1)},\ (t_{[\star j]})_{[\star i]} = (t_{[\star(i+1)]})_{[\star j]}$	

that $E \notin S$ or $E = C \in S$ and s is not a pair. Otherwise, when $E = A, C$ or AC and $E \in S$, rules ($\approx_\alpha$ **A**), ($\approx_\alpha$ **C**) or ($\approx_\alpha$ **AC**) apply. Therefore, if f is not an A, C or AC function symbol or $A, C, AC \notin S$, the behaviour of ($\approx_\alpha$ $\overline{\textbf{app}}$) and ($\approx_\alpha$ **app**) would be exactly the same. These rules define an extended calculus for *general* α-equivalence modulo A, C and AC. Other equational theories might be included similarly. Below, $\nabla \vdash s \approx_{\{A,C,AC\}} t$ denotes that s and t are α-equivalent modulo A, C and AC under the context ∇.

$$\frac{\nabla \vdash s \approx_{\{A,C,AC\}} t}{\nabla \vdash f_k^E s \approx_{\{A,C,AC\}} f_k^E t} \quad \begin{array}{l} E \notin S \text{ or} \\ E = C \text{ and } s \text{ is not a pair} \end{array} \quad (\approx_\alpha \overline{\textbf{app}})$$

Figure 5: ($\approx_\alpha$ $\overline{\textbf{app}}$)-rule for $\approx_{\{A,C,AC\}}$

$$\frac{\begin{array}{l} \nabla \vdash (f_k^A s)_{(\mathbf{1})_{f_k^A}} \approx_{\{A,C,AC\}} (f_k^A t)_{(\mathbf{1})_{f_k^A}}, \\ \nabla \vdash (f_k^A s)_{[\star\mathbf{1}]_{f_k^A}} \approx_{\{A,C,AC\}} (f_k^A t)_{[\star\mathbf{1}]_{f_k^A}} \end{array}}{\nabla \vdash f_k^A s \approx_{\{A,C,AC\}} f_k^A t} \ (\approx_\alpha \mathbf{A})$$

Figure 6: ($\approx_\alpha$ **A**)-rule for A function symbols

Rule ($\approx_\alpha$ **A**) applies when the terms compared are headed by the same A function symbol and $A \in S$. It verifies recursively if the first arguments on the left (*lhs*) and right-hand sides (*rhs*) are related by $\approx_{\{A,C,AC\}}$ as well as the result of applying the root function symbol to the respective tuples without the first argument.

Rule ($\approx_\alpha$ **C**) has two possibilities of application: for $i = 0$ (resp. $i = 1$) one must have $\nabla \vdash s_0 \approx_{\{A,C,AC\}} t_0$ and $\nabla \vdash s_1 \approx_{\{A,C,AC\}} t_1$ (resp. $\nabla \vdash s_0 \approx_{\{A,C,AC\}} t_1$ and $\nabla \vdash s_1 \approx_{\{A,C,AC\}} t_0$). The case where f_K^C is applied to a term different of a pair is considered in the ($\approx_\alpha$ $\overline{\textbf{app}}$)-rule.

$$\frac{\nabla \vdash s_0 \approx_{\{A,C,AC\}} t_i,\ \nabla \vdash s_1 \approx_{\{A,C,AC\}} t_{1-i}}{\nabla \vdash f_k^C \langle s_0, s_1 \rangle \approx_{\{A,C,AC\}} f_k^C \langle t_0, t_1 \rangle}\ i = 0,1\ (\approx_\alpha \mathbf{C})$$

Figure 7: $(\approx_\alpha \mathbf{C})$-rule for C function symbols

$$\frac{\begin{array}{l}\nabla \vdash (f_k^{AC}\, s)_{(\mathbf{1})_{f_k^{AC}}} \approx_{\{A,C,AC\}} (f_k^{AC}\, t)_{(\mathbf{i})_{f_k^{AC}}}, \\ \nabla \vdash (f_k^{AC}\, s)_{[\star\mathbf{1}]_{f_k^{AC}}} \approx_{\{A,C,AC\}} (f_k^{AC}\, t)_{[\star\mathbf{i}]_{f_k^{AC}}}\end{array}}{\nabla \vdash f_k^{AC}\, s \approx_{\{A,C,AC\}} f_k^{AC}\, t}\ AC \in S\ (\approx_\alpha \mathbf{AC})$$

Figure 8: $(\approx_\alpha \mathbf{AC})$-rule for AC function symbols

Rule $(\approx_\alpha \mathbf{AC})$ behaves similarly to rule $(\approx_\alpha \mathbf{A})$: the fundamental difference is that the first argument on the *lhs* can be compared modulo $\approx_{\{A,C,AC\}}$ with any arbitrary argument on the *rhs*. If there exists such argument, say the i^{th}, it remains to check that the terms obtained applying the function symbol to the tuples deleting the first and the i^{th} arguments to the right and to the left are related by $\approx_{\{A,C,AC\}}$.

Example 5. $\nabla \vdash f\,\langle t_1, g^{AC}\,\langle t_2, g^{AC}\langle t_3, t_4\rangle\rangle\rangle \approx_{\{A,C,AC\}} f\,\langle t_1, g^{AC}\,\langle\langle t_4, t_3\rangle, t_2\rangle\rangle$, *where g is AC, f is a function symbol that allows only α-equivalence and $AC \in S$.*

4.3. Checking $\approx_{\{A,C,AC\}}$, $\approx_{\alpha,A}$, $\approx_{\alpha,C}$ and $\approx_{\alpha,AC}$

The present formalisation adds to [17] the treatment of C-operators, which requires the analysis of an additional case in proofs of lemmas on intermediate transitivity for $\approx_{\{A,C,AC\}}$, freshness preservation under $\approx_{\{A,C,AC\}}$, equivariance, reflexivity, symmetry and transitivity of $\approx_{\{A,C,AC\}}$ and, combination of AC arguments (Lemmas 9 to 15, among others).

The following steps were performed in order to check that $\approx_{\{A,C,AC\}}$ is indeed an equivalence relation. After proving an intermediate transitivity lemma for $\approx_{\{A,C,AC\}}$ (Lemma 9), one proves *freshness preservation* and *equivariance* (Lemmas 10, 11) of $\approx_{\{A,C,AC\}}$ and then, transitivity before symmetry (Lemmas 14 and 15). By using the parameter set S on the **equiv**(S) relation and renaming superscripts of function symbols, one obtains as corollary of the soundness $\approx_{\{A,C,AC\}}$ the soundness of $\approx_{\alpha,A}$, $\approx_{\alpha,C}$ and $\approx_{\alpha,AC}$.

In addition to preservation of freshness and equivariance, the intermediate transitivity lemma (Lemma 9) is relevant to guarantee some key properties on swappings and permutations acting over $\approx_{\{A,C,AC\}}$-related terms as for instance, $\nabla \vdash t \approx_{\{A,C,AC\}} (a\,a') \cdot t' \Rightarrow \nabla \vdash (a'\,a) \cdot t \approx_{\{A,C,AC\}} t'$.

Lemma 9 (Intermediate transitivity for $\approx_{\{A,C,AC\}}$ with $\approx_\alpha$)**.** *If* $\nabla \vdash s \approx_{\{A,C,AC\}} t$ *and* $\nabla \vdash t \approx_\alpha u$ *then* $\nabla \vdash s \approx_{\{A,C,AC\}} u$.

The formalisation is obtained as follows: after generalisation of u, induction is applied on deduction rules of $\approx_{\{A,C,AC\}}$ for $\nabla \vdash s \approx_{\{A,C,AC\}} t$. Some cases require analysis over the premisse $\nabla \vdash t \approx_\alpha u$; for instance, in the case in which one has $t = \langle t_1, t_2 \rangle$, inversion is applied to obtain that $u = \langle u_1, u_2 \rangle$ with $\nabla \vdash t_1 \approx_\alpha u_1$ and $\nabla \vdash t_2 \approx_\alpha u_2$, according to the inference rule ($\approx_\alpha$ **pair**).

Lemma 10 (Freshness preservation under $\approx_{\{A,C,AC\}}$)**.** *If* $\nabla \vdash a \# s$ *and* $\nabla \vdash s \approx_{\{A,C,AC\}} t$ *then* $\nabla \vdash a \# t$.

The proof is by induction on $\approx_{\{A,C,AC\}}$, using some technical results about the freshness relation for dealing with cases related with rules ($\approx_\alpha$ [**aa**]) and ($\approx_\alpha$ [**ab**]) for the case in which s and t are abstractions.

Lemma 11 (Equivariance of $\approx_{\{A,C,AC\}}$)**.** *If* $\nabla \vdash s \approx_{\{A,C,AC\}} t$ *then* $\nabla \vdash \pi \cdot s \approx_{\{A,C,AC\}} \pi \cdot t$.

Equivariance follows by induction in the inference rules of $\approx_{\{A,C,AC\}}$. For the case of abstractions, specifically for the case of the rule ($\approx_\alpha$ [**ab**]), Lemma 9 is required; indeed, when one has $\nabla \vdash [a]s' \approx_{\{A,C,AC\}} [b]t'$, initially it is necessary to prove that $\nabla \vdash \pi \cdot s' \approx_{\{A,C,AC\}} \pi \cdot ((a\ b) \cdot t')$ and $\nabla \vdash \pi \cdot ((a\ b) \cdot t') \approx_\alpha (\pi \cdot a\ \ \pi \cdot b) \cdot (\pi \cdot t')$ and then apply that lemma to obtain $\nabla \vdash \pi \cdot s' \approx_{\{A,C,AC\}} (\pi \cdot a\ \ \pi \cdot b) \cdot (\pi \cdot t')$.

Lemma 12 (Reflexivity of $\approx_{\{A,C,AC\}}$)**.** $\nabla \vdash t \approx_{\{A,C,AC\}} t$.

Reflexivity is easily proved by induction on t. The next lemma generalises the way in which arguments used in the rule ($\approx_\alpha$ **AC**) are combined.

Lemma 13 (Combination of AC arguments)**.** *If* $\nabla \vdash t \approx_{\{A,C,AC\}} t'$ *then* $\forall_{(0<i\leq \|t\|_f)} \exists_{(0<j\leq \|t\|_f)} \nabla \vdash t_{(i)_f} \approx_{\{A,C,AC\}} t'_{(j)_f}$ *and* $\nabla \vdash t_{[\star i]_f} \approx_{\{A,C,AC\}} t'_{[\star j]_f}$.

The proof is by induction on $\|t\|_f$ using simple auxiliary lemmas and properties of the operators $\|t\|_f, t_{(i)_f}$ and $t_{[\star i]_f}$. We explain how the proof is obtained for the particular case for $i = 1$: $\nabla \vdash t \approx_{\{A,C,AC\}} t' \Rightarrow \exists_{(0<j\leq \|t'\|_f)}, \nabla \vdash t_{(1)_f} \approx_{\{A,C,AC\}} t'_{(j)_f} \ \wedge\ \nabla \vdash t_{[\star 1]_f} \approx_{\{A,C,AC\}} t'_{[\star j]}$. The complicated case happens when $\|t\|_f > 2$: after applying the auxiliary lemma for terms $f\,t$ and $f\,t'$ one obtains for some valid i_0, $\nabla \vdash t_{(1)_f} \approx_{\{A,C,AC\}} t'_{(i_0)_f}$ and $\nabla \vdash f\,t_{[\star 1]_f} \approx_{\{A,C,AC\}} f\,t'_{[\star i_0]_f}$. Notice that if $i = 1$, the result follows trivially. For $i > 1$, induction applies for the terms $t_0 = f\,t_{[\star 1]_f}$ and $t'_0 = f\,t'_{[\star i_0]_f}$ with argument $i_1 = i - 1$. Notice that the IH is given as $\forall_{(\|t_0\|_f < \|t\|_f,\, t'_0,\, 0<i_1\leq \|t_0\|_f)} \exists j_1, \nabla \vdash t_{0(i_1)_f} \approx_{\{A,C,AC\}} t'_{0(j_1)_f}$ and $\nabla \vdash$

$t_{0[\star i_1]_f} \approx_{\{A,C,AC\}} t'_{0[\star j_1]_f}$. Then, applying IH, a witness j is obtained such that, with the pre-conditions: $\|f\, t_{[\star 1]_f}\|_f < \|t\|_f$ and $\nabla \vdash f\, t_{[\star 1]_f} \approx_{\{A,C,AC\}} f\, t'_{[\star i_0]_f}$, one obtains $\nabla \vdash f\, t_{(i)_f} \approx_{\{A,C,AC\}} f\, t'_{(j)_f}$ and $\nabla \vdash f\, t_{[\star(i)]_f} \approx_{\{A,C,AC\}} f\, t'_{[\star j]_f}$. The first pre-condition is solved by an application of the definition of $\|\_\|$ and an auxiliary lemma for the operators $\|t\|_f$ and $t_{[\star i]_f}$. The second is exactly the assumption. Then one just needs to consider two cases: $i_0 \leq j_1$ or $i_0 > j_1$. One instantiates j respectively as j_1+1 or j_1 and concludes using properties of the operators $\|t\|_f, t_{(i)_f}$ and $t_{[\star i]_f}$.

Lemma 14 (Transitivity of $\approx_{\{A,C,AC\}}$)**.** *If* $\nabla \vdash t_1 \approx_{\{A,C,AC\}} t_2$ *and* $\nabla \vdash t_2 \approx_{\{A,C,AC\}} t_3$ *then* $\nabla \vdash t_1 \approx_{\{A,C,AC\}} t_3$.

The formalisation is by induction on the size of the term t_1. The terms t_2 and t_3 are generalised, and inversions from the equational inference rules are applied to both $\nabla \vdash t_1 \approx_{\{A,C,AC\}} t_2$ and $\nabla \vdash t_2 \approx_{\{A,C,AC\}} t_3$. The difficult cases are those of rules ($\approx_\alpha$ [**ab**]) and ($\approx_\alpha$ **A**) or ($\approx_\alpha$ **AC**). For ($\approx_\alpha$ [**ab**]), an interesting subcase is when $a \neq a' \neq a'_0 \neq a$: the premisses are $\nabla \vdash t \approx_{\{A,C,AC\}} (a\, a') \cdot t' \ \wedge\ \nabla \vdash a \# t'$ and $\nabla \vdash t' \approx_{\{A,C,AC\}} (a'\, a'_0) \cdot t'_0 \ \wedge\ \nabla \vdash a'_0 \# t'_0$, the IH is given as $\forall_{(s_1,s_2,s_3)}, |s_1| < |t| \wedge (\nabla \vdash s_1 \approx_{\{A,C,AC\}} s_2 \wedge \nabla \vdash s_2 \approx_{\{A,C,AC\}} s_3) \Rightarrow \nabla \vdash s_1 \approx_{\{A,C,AC\}} s_3$, and one should conclude that $\nabla \vdash [a]t \approx_{\{A,C,AC\}} [a'_0]t'_0$. Applying ($\approx_\alpha$ [**ab**]) it remains to prove that $\nabla \vdash a \# t'_0$ and $\nabla \vdash t \approx_{\{A,C,AC\}} (a\, a'_0) \cdot t'_0$. The former is obtained by freshness preservation, and the latter by IH with application of Lemma 9, equivariance and freshness preservation.

In the case of rules ($\approx_\alpha$ **A**) or ($\approx_\alpha$ **AC**), the following proof context is reached at some point of the formalisation, where for the case of ($\approx_\alpha$ **A**), the indices i and i_0 are equal to 1: the premisses are $\nabla \vdash t_{(1)_{f_k^E}} \approx_{\{A,C,AC\}} t'_{(i)_{f_k^E}} \wedge \nabla \vdash f_k^E\, t_{[\star 1]_{f_k^E}} \approx_{\{A,C,AC\}} f_k^E\, t'_{[\star i]_{f_k^E}}$, and $\nabla \vdash t'_{(1)_{f_k^E}} \approx_{\{A,C,AC\}} t'_{0(i_0)_{f_k^E}} \wedge \nabla \vdash f_k^E\, t'_{[\star 1]_{f_k^E}} \approx_{\{A,C,AC\}} f_k^E\, t'_{0[\star i_0]_{f_k^E}}$, the IH is given by $\forall_{(s_1,s_2,s_3)}, |s_1| < |f_k^E\, t| \wedge (\nabla \vdash s_1 \approx_{\{A,C,AC\}} s_2 \wedge \nabla \vdash s_2 \approx_{\{A,C,AC\}} s_3) \Rightarrow \nabla \vdash s_1 \approx_{\{A,C,AC\}} s_3$, and one should conclude that $\nabla \vdash f_k^E\, t \approx_{\{A,C,AC\}} f_k^E\, t'_0$. Applying ($\approx_\alpha$ **A**) and the IH one concludes easily for the case in which $E = A$. When $E = AC$ one uses the Lemma 13 and the second premise above, obtaining a third premise: $\exists i_1, \nabla \vdash t'_{(i)_{f_k^E}} \approx_{\{A,C,AC\}} t'_{0(i_1)_{f_k^E}} \wedge \nabla \vdash t'_{[\star i]_{f_k^E}} \approx_{\{A,C,AC\}} t'_{0[\star i_1]_{f_k^E}}$. Then, applying the ($\approx_\alpha$ **AC**) rule instantiated with i_1. The resulting subgoals are $\nabla \vdash t_{(1)_{f_k^E}} \approx_{\{A,C,AC\}} t'_{0(i_1)_{f_k^E}}$ and $\nabla \vdash f_k^E\, t_{[\star 1]_{f_k^E}} \approx_{\{A,C,AC\}} f_k^E\, t'_{0[\star i_1]_{f_k^E}}$, and from the first and third premises above, both subgoals are solved by application of IH.

Lemma 15 (Symmetry of $\approx_{\{A,C,AC\}}$)**.** *If* $\nabla \vdash t \approx_{\{A,C,AC\}} t'$ *then* $\nabla \vdash t' \approx_{\{A,C,AC\}} t$.

Symmetry is easily formalised by induction on $\approx_{\{A,C,AC\}}$ applying lemmas 9, 12 and 14, freshness preservation and equivariance.

In particular, the use of the Lemma 14 is crucial: in the $(\approx_\alpha [\mathbf{ab}])$ case one should prove that $\nabla \vdash [b]t' \approx_{\{A,C,AC\}} [a]t$ having as hypotheses $\nabla \vdash t \approx_{\{A,C,AC\}} (a\,b)\cdot t'$ and $\nabla \vdash a \# t'$, with IH $\nabla \vdash (a\,b)\cdot t' \approx_{\{A,C,AC\}} t$. Then, Lemma 14 is applied twice instantiating t_2 as $(a\,b)\cdot t$ and as $(a\,b)\oplus(a\,b)\cdot t'$, this allows the use of Lemmas 9 (with properties of $\approx_\alpha$) and equivariance to conclude.

The following corollary is used to derive, from Lemmas 12, 14 and 15 the proofs that $\approx_{\alpha,A}$, $\approx_{\alpha,C}$, $\approx_{\alpha,AC}$ and $\approx_{\{A,C,AC\}}$ are indeed equivalence relations. Remember the parameterisation used in the specification, in which **equiv** with arguments sets of indices $\{0\}$, $\{1\}$, $\{2\}$ and $\{0,1,2\}$ correspond respectively to $\approx_{\alpha,A}$, $\approx_{\alpha,AC}$, $\approx_{\alpha,C}$ and $\approx_{\{A,C,AC\}}$.

Corollary 1. *For $S \subseteq \{0,1,2\}$, $\mathbf{equiv}(S)$ is also an equivalence relation.*

The formalisation is obtained by the manipulation of the superscripts in $S^{-1} = \{0,1,2\} - S$. For a general equivalence problem $\mathbf{equiv}(S)(\nabla, t_1, t_2)$, one replaces all superscripts of the operators in the terms t_1 and t_2 inside the set S^{-1} by new ones that neither belong to $\{0,1,2\}$ nor occur in t_1 and t_2 obtaining respectively t_1' and t_2'. Then, by induction on the inference rules for **equiv**, one easily proves that $\mathbf{equiv}(S)(\nabla, t_1, t_2) \Leftrightarrow \mathbf{equiv}(S)(\nabla, t_1', t_2') \Leftrightarrow \mathbf{equiv}(\{0,1,2\})(\nabla, t_1', t_2')$. Thus, using that $\mathbf{equiv}(\{0,1,2\})$ is an equivalence relation one concludes.

4.4. Formalisation Details

The key code fragment of the formalisation is the inductive definition **equiv** in Figure 9 (available in the file `Equiv.v` of the specification). This definition uses notations and operators given in Figures 1 and 4, and specifies a relation that has type Context → **term** → **term** → `Prop` and a set of naturals S as parameter. This definition includes specific rules for each constructor of the nominal syntax, and a signature that may contain A, C and AC function symbols according to S.

The inference rules $(\approx_\alpha \langle\rangle)$, $(\approx_\alpha \mathbf{atom})$, $(\approx_\alpha \mathbf{app})$, $(\approx_\alpha [\mathbf{aa}])$, $(\approx_\alpha [\mathbf{ab}])$, $(\approx_\alpha \mathbf{var})$, $(\approx_\alpha \mathbf{pair})$, $(\approx_\alpha \mathbf{A})$ and $(\approx_\alpha \mathbf{AC})$ are specified, respectively, by the following constructors of **equiv**: equiv_Ut; equiv_At; equiv_Fc; equiv_Ab_1; equiv_Ab_2; equiv_Su; equiv_Pr; equiv_A and equiv_AC. And additionally, the two cases of rule $(\approx_\alpha \mathbf{C})$ are specified by equiv_C1 and equiv_C2.

The constructor equiv_Ut (resp. equiv_At) express that $\langle\rangle$ (resp. $\bar{a}$) is related with itself, for any S and any C. In equiv_Fc, Fc E n t is related to Fc E n t', if E does not belong to S, that means one is not dealing with an A, AC or C function symbol, or if $E = 2$, that means one is dealing with a C function symbol which is not applied to a pair $((\neg$ is_Pr $t) \vee (\neg$ is_Pr $t'))$. Notice that, equiv_Fc was specified to cover the cases in which rules for A, C or an AC function symbols do not apply.

```
Inductive equiv (S : set nat): Context → term → term → Prop :=

 | equiv_Ut : ∀ C, equiv S C (<<>>) (<<>>)

 | equiv_At : ∀ C a, equiv S C (%a) (%a)

 | equiv_Pr : ∀ C t1 t2 t1' t2',
   (equiv S C t1 t1') → (equiv S C t2 t2') → equiv S C (<| t1, t2 |>) (<| t1',t2' |>)

 | equiv_Fc : ∀ E n t t' C, (¬ set_In E S ∨ (E = 2 ∧ ((¬ is_Pr t) ∨ (¬ is_Pr t')))) →
   (equiv S C t t') → equiv S C (Fc E n t) (Fc E n t')

 | equiv_Ab_1 : ∀ C a t t', (equiv S C t t') → equiv S C ([a]^t) ([a]^t')

 | equiv_Ab_2 : ∀ C a a' t t', a ≠ a' →
   (equiv S C t (|[(a,a']|@ t')) → C ⊢ a # t' → equiv S C ([a]^t) ([a']^t')

 | equiv_Su : ∀ (C: Context) p p' (X: Var), (∀ a, (In_ds p p' a) → set_In (a, X) C) →
   equiv S C (p|.X) (p'|.X)

 | equiv_A : set_In 0 S → ∀ n t t' C,
   (equiv S C (TPith 1 (Fc 0 n t) 0 n) (TPith 1 (Fc 0 n t') 0 n)) →
   (equiv S C (TPithdel 1 (Fc 0 n t) 0 n) (TPithdel 1 (Fc 0 n t') 0 n)) →
   (equiv S C (Fc 0 n t) (Fc 0 n t'))

 | equiv_AC : set_In 1 S → ∀ n t t' i C,
   (equiv S C (TPith 1 (Fc 1 n t) 1 n) (TPith i (Fc 1 n t') 1 n)) →
   (equiv S C (TPithdel 1 (Fc 1 n t) 1 n) (TPithdel i (Fc 1 n t') 1 n)) →
   (equiv S C (Fc 1 n t) (Fc 1 n t'))

| equiv_C1 : set_In 2 S → ∀ n s0 s1 t0 t1 C,
   (equiv S C s0 t0) → (equiv S C s1 t1) →
   (equiv S C (Fc 2 n (<| s0, s1 |>)) (Fc 2 n (<| t0, t1 |>)))

| equiv_C2 : set_In 2 S → ∀ n s0 s1 t0 t1 C,
   (equiv S C s0 t1) → (equiv S C s1 t0) →
   (equiv S C (Fc 2 n (<| s0, s1 |>)) (Fc 2 n (<| t0, t1 |>))) .
```

Figure 9: Specification of **equiv**

```
Inductive fresh : Context → Atom →term → Prop :=
 | fresh_Ut : ∀ C a, fresh C a Ut
 | fresh_Pr : ∀ C a t1 t2, (fresh C a t1) → (fresh C a t2) →
   (fresh C a (<| t1 , t2 |>))
 | fresh_Fc : ∀ C a E n t, (fresh C a t) → (fresh C a (Fc E n t))
 | fresh_Ab_1 : ∀ C a t, fresh C a (Ab a t)
 | fresh_Ab_2 : ∀ C a a' t, a ≠ a' →
   (fresh C a t) → (fresh C a (Ab a' t))
 | fresh_At : ∀ C a a', a ≠ a' → (fresh C a (At a'))
 | fresh_Su : ∀ C p a X, set_In ((!p $ a), X)) C →
   fresh C a (Su p X) .
```

Figure 10: Specification of **fresh**

The constructor equiv_Ab_2 relates `[`a`]^`t to `[`a'`]^`t', considering S and a freshness context C, whenever the atoms are different (i.e., $a \neq a'$), t is related with `|[(`a`,`a'` ]| @ `t' (an application of swapping $(a\,a')$ to t') and a is fresh in t' in the context C, which uses notation $C \vdash a$ `#` t' in the specification. The inductive definition **fresh** in Figure 10, specifies the freshness relation given in Figure 2 (available in the file `Fresh.v` of the specification).

The constructor equiv_AC relates Fc 1 n t to Fc 1 n t', given S and C, whenever 1 belongs to S (otherwise equiv_Fc is applied) and there exists an i such that TPith 1 (Fc 1 n t) 1 n is related to TPith i (Fc 1 n t') 1 n and TPithdel 1 (Fc 1 n t) 1 n is related to TPithdel i (Fc 1 n t') 1 n.

Since **equiv** is defined inductively, Coq automatically gives the induction schemes that are applied in induction and case analysis proofs over **equiv**. This inductive approach that is natural in Coq, is the main difference between the present Coq specification and the PVS formalisation given in [5]. In the latter, the authors used a functional (recursive) specification for the treatment of α-unification which easily can be adapted just for the case of α-equality check. From a pragmatic point of view, induction and recursion are possible in both proof assistants and the choice of one or the other style of specification is motivated by a preference in the style of formalisation and not in restrictions inherent to the deductive power of the proof assistant. Despite this fact, it should be stressed that recursive definitions are closer to functional specifications than inductive definitions. In Subsection 5.2 an alternative recursive definition in Coq is presented that has been proved to be equivalent to the relation $\approx_{\{A,C,AC\}}$, and from which OCaml executable code has

been automatically extracted.

5. Upper bounds for general $\approx_{\alpha,A}$, $\approx_{\alpha,C}$, $\approx_{\alpha,AC}$ and $\approx_{\{A,C,AC\}}$ problems

This section is concerned with the problem of checking the validity of α-equivalence constraints in the presence of A, C and AC function symbols, by applying simplification rules.

For example, using the simplification rules given in [7], a constraint of the form $[a]X \approx_\alpha [b]X$ reduces to the set of constraints $a \# X$, $b \# X$; therefore, $a \# X, b \# X \vdash [a]X \approx_\alpha [b]X$. Similarly, assuming $+$ is an AC function symbol, the equality $\nabla \vdash +\langle s, +\langle t, [a]X\rangle\rangle \approx_{\alpha,AC} +\langle +\langle [b]X, s\rangle, t\rangle$ holds whenever the freshness constraints $a \# X$, $b \# X$ belong to ∇. Equational problems will be written as pairs $\langle \nabla, P\rangle$, where ∇ is a set of freshness constraints and P a set of equations. For simplicity, when no confusion arises brackets will be omitted.

5.1. A naive implementation

An algorithm to check a problem $\langle \nabla, P\rangle$ modulo A/C/AC is defined by the recursive function `Check` given in Algorithm 1. This algorithm simply distinguishes the cases that should be considered to deal with A/C/AC function symbols.

Example 6. *Assuming $\nabla = \{a \# X, b \# X\}$ and using the algorithm, where g is a syntactic function symbol, it follows that*

$$\begin{aligned}
&\nabla, \{[a]g\langle \overline{a}, X\rangle \approx [b]g\langle \overline{b}, X\rangle\} \Longrightarrow_{Line\,12} \nabla, \{g\langle \overline{a}, X\rangle \approx (a\,b)\cdot g\langle \overline{b}, X\rangle\}\\
&= \nabla, \{g\langle \overline{a}, X\rangle \approx g\langle \overline{a}, (a\,b).X\rangle\} \Longrightarrow_{Line\,38} \nabla, \{\langle \overline{a}, X\rangle \approx \langle \overline{a}, (a\,b)X\rangle\}\\
&\Longrightarrow_{Line\,8} \nabla, \{\overline{a} \approx \overline{a}, X \approx (a\,b).X\} \Longrightarrow_{Line\,6,16} \nabla, \emptyset \Longrightarrow_{Line\,2} \top
\end{aligned}$$

Example 7. *Consider the problem $\langle \emptyset, \{f_k^A\langle \overline{a}, \langle \overline{b}, [a]\overline{a}\rangle\rangle \approx f_k^A\langle \langle \overline{a}, \overline{b}\rangle, [b]\overline{b}\rangle\}\rangle$.*

$$\begin{aligned}
&\emptyset, \{f_k^A\langle \overline{a}, \langle \overline{b}, [a]\overline{a}\rangle\rangle \approx f_k^A\langle \langle \overline{a}, \overline{b}\rangle, [b]\overline{b}\rangle\}\\
&\Longrightarrow_{Line\,19,21} \emptyset, \{f_k^A\langle \overline{b}, [a]\overline{a}\rangle \approx f_k^A\langle \overline{b}, [b]\overline{b}\rangle\}, \quad \textit{since } \texttt{Check}(\emptyset, \overline{a} \approx \overline{a}) \textit{ (Line 20)}\\
&\Longrightarrow_{Line\,19,21} \emptyset, \{f_k^A[a]\overline{a} \approx f_k^A[b]\overline{b}\}, \quad \textit{since } \texttt{Check}(\emptyset, \overline{b} \approx \overline{b}) \textit{ (Line 20)}\\
&\Longrightarrow_{Line\,19,21} \emptyset, \{\langle\rangle \approx \langle\rangle\}, \quad \textit{since } \texttt{Check}(\emptyset, [a]\overline{a} \approx [b]\overline{b}) \textit{ (Line 20)}\\
&\Longrightarrow_{Line\,5,2} \top
\end{aligned}$$

Notice that, in the third step above, one calls $\texttt{Check}(\emptyset, \langle\rangle \approx \langle\rangle)$ *since* $(f_k^A[a]\overline{a})_{[\star 1]_{f_k^A}} = (f_k^A[b]\overline{b})_{[\star 1]_{f_k^A}} = \langle\rangle$ *(see Figure 4).*

Algorithm 1 Checking α-equivalence modulo A, C and AC

1: **function** $\texttt{Check}(\nabla, P)$

2: **if** $P = \emptyset$ **then** $\top$

3: **else let** $s \approx t \in P$ and $P' = P \setminus \{s \approx t\}$ **in**

4: **case** $\boldsymbol{s \approx t}$ **of**

5: $\langle\rangle \approx \langle\rangle$ **:** $\texttt{Check}(\nabla, P')$ // rule ($\approx_\alpha \langle\rangle$)

6: $\bar{a} \approx \bar{a}$ **:** $\texttt{Check}(\nabla, P')$ // rule ($\approx_\alpha$ **atom**)

7: $\langle s_1, s_2\rangle \approx \langle t_1, t_2\rangle$ **:**

8: $\texttt{Check}(\nabla, \{s_1 \approx t_1,\ s_2 \approx t_2\} \cup P')$ // rule ($\approx_\alpha$ **pair**)

9: $[a]s' \approx [a]t'$ **:** $\texttt{Check}(\nabla, \{s' \approx t'\} \cup P')$ // rule ($\approx_\alpha$ [**aa**])

10: $[a]s' \approx [b]t'$ **:** // rule ($\approx_\alpha$ [**ab**])

11: **if** $\nabla \vdash a \# t'$ **then**

12: $\texttt{Check}(\nabla, \{s' \approx (a\,b) \cdot t'\} \cup P')$

13: **else** $\bot$

14: **end if**

15: $\pi.X \approx \pi'.X$ **:** // rule ($\approx_\alpha$ **var**)

16: **if** For all $a \in ds(\pi, \pi'), a \# X \in \nabla$ **then** $\texttt{Check}(\nabla, P')$

17: **else** $\bot$

18: **end if**

19: $f_k^A s' \approx f_k^A t'$ **:** // rule ($\approx_\alpha$ **A**)

20: **if** $\texttt{Check}(\nabla, \{(f_k^A s')_{(1)_{f_k^A}} \approx (f_k^A t')_{(1)_{f_k^A}}\})$ **then**

21: **if** $\texttt{Check}(\nabla, \{(f_k^A\ s)_{[\star 1]_{f_k^A}} \approx (f_k^A\ t)_{[\star 1]_{f_k^A}}\})$ **then** $\texttt{Check}(\nabla, P')$

22: **else** $\bot$

23: **end if**

24: **else** $\bot$

25: **end if**

26: $f_k^C \langle s_0, s_1\rangle \approx f_k^C \langle t_0, t_1\rangle$ **:** // rule ($\approx_\alpha$ **C**)

27: **if** $\texttt{Check}(\nabla, \{s_0 \approx t_0,\ s_1 \approx t_1\})$ **then** $\texttt{Check}(\nabla, P')$

28: **else**

29: **if** $\texttt{Check}(\nabla, \{s_0 \approx t_1,\ s_1 \approx t_0\})$ **then** $\texttt{Check}(\nabla, P')$

30: **else** $\bot$

31: **end if**

32: **end if**

33: $f_k^{AC} s' \approx f_k^{AC} t'$ **:** // rule ($\approx_\alpha$ **AC**)

34: **let** $\texttt{Branch}(i) :=$

35: **if** $\texttt{Check}(\nabla, \{(f_k^{AC}\ s)_{(1)_{f_k^{AC}}} \approx (f_k^{AC}\ t)_{(i)_{f_k^{AC}}}\})$ **then**

36: $\texttt{Check}(\nabla, \{(f_k^{AC}\ s)_{[\star 1]_{f_k^{AC}}}) \approx (f_k^{AC}\ t)_{[\star i]_{f_k^{AC}}})\})$

37: **else** $\bot$

38: **end if in**

39: **if** $\texttt{Iter}(\texttt{Branch}, 1, \|f_k^{AC}\ t\|)$ **then** $\texttt{Check}(\nabla, P')$

40: **else** $\bot$

41: **end if**

42: $f_k^E s' \approx f_k^E t'$ **:** $\texttt{Check}(\nabla, \{s' \approx t'\} \cup P')$ // rule ($\approx_\alpha \overline{\textbf{app}}$)

43: $\_\_$ **:** $\bot$ // otherwise

44: **end if**

45: **end function**

Example 8. *Consider the problem* $\langle \emptyset, \{f_k^C\langle \bar{b}, [a]\bar{a}\rangle \approx f_k^C\langle [b]\bar{b}, \bar{b}\rangle\}\rangle$.

$$\begin{aligned}
&\emptyset, \{f_k^C\langle \bar{b}, [a]\bar{a}\rangle \approx f_k^C\langle [b]\bar{b}, \bar{b}\rangle\}\\
&\Longrightarrow_{Line\,29} \emptyset, \{\bar{b} \approx \bar{b},\ [a]\bar{a} \approx [b]\bar{b}\},\\
&\qquad\qquad\qquad\qquad \textit{since } \texttt{Check}(\emptyset, \{\bar{b} \approx [b]\bar{b},\ [a]\bar{a} \approx \bar{b}\}) = \bot \textit{ (L. 27)}\\
&\Longrightarrow_{Line\,6} \emptyset, \{[a]\bar{a} \approx [b]\bar{b}\} \Longrightarrow_{Line\,10,12,2} \top
\end{aligned}$$

Lines 33 to 41 in Algorithm 1 deal with the case of equations headed by AC-function symbols. The algorithm checks equality of the first argument on the *lhs* of the equation with the first, second, third, etc. of the *rhs* until this check succeeds and then recursively checks equality of the whole term obtained by eliminating the first argument on the *lhs* and the successful i^{th} argument on the *rhs*; otherwise, the search continues recursively increasing i^{th} until it exceeds the number of arguments of the heading function symbol in $f_k^{AC}s'$, and the check fails. This is specified in Coq through a simple recursive implementation of an iteration function `Iter`.

Example 9. *Consider the problem* $\langle \nabla, \{f_k^{AC}([a]a, \pi.X) \approx f_k^{AC}(\pi'.X, [b]b)\}\rangle$ *and assume that* $ds(\pi, \pi')\#X \subseteq \nabla$. *The algorithm* `Check` *will call* $\texttt{Check}_{\texttt{AC}}$ *proceeding as follows:*

$$\begin{aligned}
&\nabla, \{f_k^{AC}([a]\bar{a}, \pi.X) \approx f_k^{AC}(\pi'.X, [b]\bar{b})\}\\
&\Longrightarrow_{Line\,36} \texttt{Check}(\nabla, f_k^{AC}[a]\bar{a} \approx f_k^{AC}[b]\bar{b})\\
&\qquad\qquad\qquad\qquad \textit{since } \texttt{Check}(\nabla, \{[a]\bar{a} \approx \pi'.X\}) = \bot \textit{ (Line 35, 43)}\\
&\Longrightarrow_{Line\,36} \nabla, \{f_k^{AC}\ \pi.X \approx f_k^{AC}\ \pi'.X\},\\
&\qquad\qquad\qquad\qquad \textit{since } \texttt{Check}(\nabla, \{[a]\bar{a} \approx [b]\bar{b}\}) = \top \textit{ (Line 35, 10, 6, 2)}\\
&\Longrightarrow_{Line\,35} \nabla, \{\langle\rangle \approx \langle\rangle\},\\
&\qquad\qquad\qquad\qquad \textit{since } \texttt{Check}(\nabla, \pi.X \approx \pi'.X) = \top \textit{ (Line 15)}\\
&\Longrightarrow_{Line\,5,2} \top
\end{aligned}$$

Note that the proposed algorithm can check validity of α-equivalence constraints modulo A and/or C and/or AC ($\approx_{\{A,C,AC\}}$) with multiple occurrences of function symbols, some that might be A and some C and some other AC, all at once. This is due to the fact that there are no interactions between A, C, and AC symbols since distributive properties are not considered.

5.2. Extraction of a naive algorithm from the Coq specification

To obtain executable code from the inductive definition of **equiv**$(\{0, 1, 2\})$ (see Subsection 4.4), an equivalent recursive function, called equiv_rec, has been specified.

This recursive function applies the α, A, C and AC equivalence inference rules. Since the applications of rules ($\approx_\alpha$ **A**) and ($\approx_\alpha$ **AC**) require recursive applications of inference rules to equations over terms that are not subterms of the input equation, the standard `Fixpoint` definition of Coq is not applicable. Thus, a more powerful recursive combinator was used that allows well-founded recursion.

Also, the recursive calls generated by the application of rule ($\approx_\alpha$ **AC**) were specified through an auxiliary (recursively defined) iteration operator that makes the application of the fixed point Coq mechanisms difficult.

For the verification of $\mathsf{equiv\_rec}$, the techniques given by Margin and Sozeau [20] were adopted. The applied strategy uses a new type of definition named `Equations` that allows the simultaneous use of well-founded and iterative recursion, automatically generating the simplification lemmas required in the inductive formalisation of the correctness of $\mathsf{equiv\_rec}$. Other techniques are available in Coq for defining more complex recursive functions, such as the `Program Fixpoint` definition [21]. This allows the specification of functions with well-founded and iterative recursion, but the simplification lemmas are not automatically generated.

Another way of building recursive functions from inductive definitions in Coq is proposed in recent work by Larchey-Wendling and Monin [22]. The strategy consists in defining first the graph of the inductive definition; then, one proves termination, functionality and totality (over a specific domain) of the graph. This strategy also allows the use of Coq code extraction, but the process of constructing the recursive definition is not as straightforward as the `Equation` approach applied in this work.

The following lemma (formalised in Coq) verifies the recursive function $\mathsf{equiv\_rec}$ stating its equivalence to the inductive definition $\approx_{\{A,C,AC\}}$.

Lemma 16 (Correctness of $\mathsf{equiv\_rec}$)**.** $\nabla \vdash s \approx_{\{A,C,AC\}} t$ *if and only if* $(\mathsf{equiv\_rec}\ \nabla\ s\ t = \mathsf{true})$.

Proof. Necessity is proved by induction on the derivation rules of $\nabla \vdash s \approx_{\{A,C,AC\}} t$. Each case uses a previous result based on an automatically generated simplification lemma for $\mathsf{equiv\_rec}$. For instance, for the case of rule ($\approx_\alpha$ [**ab**]) the hypotheses are $a \neq b$, $\nabla \vdash u \approx_{\{A,C,AC\}} (a\,b) \cdot v$ and $\nabla \vdash a\#v$, and IH is given by $(\mathsf{equiv\_rec}\ \nabla\ u\ ((a\,b) \cdot v) = \mathsf{true})$. A previous result allows to rewrite the objective $(\mathsf{equiv\_rec}\ \nabla\ ([a]u)\ ([b]v) = \mathsf{true})$ to $((\mathsf{fresh\_rec}\ \nabla\ a\ v = \mathsf{true}) \wedge (\mathsf{equiv\_rec}\ \nabla\ u\ ((a\,b) \cdot v) = \mathsf{true}))$. After rewriting IH, one concludes using a previous correctness lemma for $\mathsf{fresh\_rec}$ which states that $\nabla \vdash a\#v$ if and only if $(\mathsf{fresh\_rec}\ \nabla\ a\ v = \mathsf{true})$.

Sufficiency is reached by induction on the size of s and case analysis over s and t. One of the non-trivial cases is when both terms s and t are headed by the same AC function symbol f. In this case the hypotheses are $l = |u| + 1$

and $(\mathsf{equiv\_rec}\,\nabla\,(f\,u)\;(f\,v) = \mathsf{true})$, and IH is given by $\forall m, m < l \Rightarrow \forall u_0, \forall v_0$, $(\mathsf{equiv\_rec}\,\nabla\,u_0\;v_0 = \mathsf{true}) \Rightarrow \nabla \vdash s_0 \approx_{\{A,C,AC\}} v_0$. A previous result allows rewriting the premisse $(\mathsf{equiv\_rec}\,\nabla\,(f\,u)\;(f\,v) = \mathsf{true})$ to $\exists i$, $(\mathsf{equiv\_rec}\,\nabla\,u_{(1)}\;v_{(i)} = \mathsf{true}) \wedge$ $(\mathsf{equiv\_rec}\;\nabla\,(f\,u)_{[\star 1]}\;(f\,v)_{[\star i]} = \mathsf{true})$. Splitting this conjunction and applying IH in both generated premisses results in two new hypotheses $\nabla \vdash u_{(1)} \approx_{\{A,C,AC\}} v_{(i)}$ and $\nabla \vdash (f\,u)_{[\star 1]} \approx_{\{A,C,AC\}} (f\,v)_{[\star i]}$. Notice that in the applications of IH the condition $m < l$ needs to be verified through basic arithmetic properties of the operators $|\_|$, $\|\_\|$, $\_(\_)$ and $\_[\star\_]$. Then, one concludes with an application of rule $(\approx_\alpha \mathbf{AC})$. □

Executable OCaml code was automatically extracted from equiv_rec, using the built-in code extraction mechanism of Coq. The generated code is available as the file `Impl/Original_Generated_Equiv.ml` inside the specification folder.

The extracted code uses Coq naturals to represent atoms, variables and indices of function symbols: n is represented as n applications of the successor constructor `S` to zero `0`. For execution tests (see Section 5.3), an adjusted version of the generated code that just replaces Coq naturals by OCaml integers, was used. The adjusted code is available as the file `Impl/Adjusted_Generated_Equiv.ml`.

In addition to the extracted naive algorithm, a manually generated one was implemented that essentially improves the representation of terms by flattening arguments of A and AC function symbols, and by a simpler analysis for the AC case than the one given by the Algorithm 1. By flattening terms, application of the selection and deletion operators, $\_\,(\_)_f$ and $\_\,[\star\_]_f$, over arguments of A and AC operators is avoided and arguments of these operator are then sequentially analysed.

The improved analysis of the AC case is inspired by the translation to the problem of finding a perfect matching in a bipartite graph given in [19] initially proposed for solving AC-matching. For a given equational problem over terms headed by an AC function symbol, a graph whose vertices are labelled by the arguments of the AC function in the *lhs* and *rhs* of the equation is built. There is an edge between two vertices labelled by arguments in opposite sides of the equation if they match. A perfect matching in the bipartite graph is a solution for the initial problem. In the case of AC-equational check, for solving the flattened equational problem $f(s_1, \ldots, s_k) \approx f(t_1, \ldots, t_k)$, if the answer is positive, and if s_i is known to be equivalent to t_l and t_j, there should be another *lhs* argument s_m that is also equivalent to these three arguments. Thus, the improved implementation essentially searches imperatively for *rhs* arguments that are equivalent to the first, second and so on *lhs* arguments (see the complexity analysis given in Theorem 1). This implementation is available as files `Impl/Basics.ml` and `Impl/Improved_Equiv.ml`.

Searching for more efficient implementations is an interesting subject of further

research, but not the objective of this paper.

5.3. Execution tests

Experiments were performed with the extracted and improved algorithms, over an iMAC server with 16GB of RAM and with a processor Intel Xeon CPU, model W3530 2.80GHz, providing randomly recursively generated ground equational problems as inputs. Terms were generated using only tuples with arguments associated to the right as arguments for A and AC function symbols. Also, subterms headed by an associative function symbol, say either f_k^A or f_k^{AC}, do not have arguments headed by the same function symbol. For example, $f_0^A\langle \bar{a}, \langle \bar{b}, \langle \bar{c}, \langle \bar{d}, \bar{e}\rangle\rangle\rangle\rangle$ and $f_0^{AC}\langle f_0^A\langle \bar{a}, \bar{b}\rangle, f_4^A\langle \bar{c}, f_0^{AC}\langle \bar{d}, \bar{e}\rangle\rangle\rangle$ are in this class of terms. Although terms generated in this manner mitigate the required effort for manipulation of arguments of associative operators, it should be stressed that the adequate data structure to deal with flattened arguments of these operators should allow random access as arrays and sequences do. Also, having only ground terms mitigates the negative effects of inefficient procedures for dealing with permutation operations, such as queries about their support, inversion and composition, which are used in the extracted algorithm for application of rule ($\approx_\alpha$ **var**).

The number of different syntactic, A, C and AC symbols were restricted to ten (each class), and atoms were chosen among a set of ten thousand. In the randomly recursive generation of an equational problem, whenever abstractions are generated as subterms, the choice of different atoms in the *lhs* and in the *rhs* of the equation is enforced. This strategy was adopted to prioritise the use of rule ($\approx_\alpha$ [**ab**]) against ($\approx_\alpha$ [**aa**]), since the latter has lower cost. In this case, to guarantee that the equality checking results in `true`, avoiding collisions and ensuring the condition $\nabla \vdash a \# t'$ of the rule ($\approx_\alpha$ [**ab**]), a list of used atoms is kept that are not allowed to occur in the body of the abstractions.

Four different sets of input problems were generated. The first one, uses only syntactic function symbols; the second uses also A symbols; the third uses syntactic, A and C symbols; and, the fourth allows all four kinds of symbols. For each set, problems with positive answer of sizes from 100 to 10000, with intervals of length 100, were generated; for each size twenty five different problems were generated. Each problem was tested with both, the extracted and the improved implementations. For the improved algorithm inputs were translated by representing tuples as lists. The cost of this syntactic translation was not considered, but of course the time required for the flattening operation was considered in the evaluation. This operation consists in the elimination of nested occurrences of A and AC function symbols. Time performance of the experiments is given in Figures 11 and 12. Plots in each row correspond to tests with the same set of inputs. Left and right plots correspond respectively to experiments with the extracted and the improved implementations.

These figures plot also the regressions computed using the generalised additive model (GAM) generated using the `ggplot2` library of R.

For all sets of inputs the performance of the improved implementation was better than the performance of the extracted implementation. As expected, from the required uniformity of known worst case inputs, which even for α-syntactic problems will result in exponential running time for the extracted algorithm, in all cases it could be observed that only a few isolated cases present running time much higher than the regression curve. The α-syntactic case (Figure 11, row 1) shows a linear behaviour for both implementations, being used the same scale in the left and right plots. Notice that the improved implementation was, approximately, 15% faster than the extracted one. This can be explained by the fact that the recursive calls in nested tuples are more time consuming than operating on more efficient data structures, such as lists, used to represent arguments of function symbols. Adding A and C-function symbols (respectively, Figure 11, row 2 and Figure 12, row 1) increases the running time as expected, but the relative behaviour is very similar. In this case the performance of the improved implementation was approximately thirty times faster than the performance of the extracted algorithm. Notice that, in the α-A and α-A-C plots the scales of the time-axis on the right are, respectively, 33.3 and 37.5 times bigger than on the left. This could be explained since the bottleneck of the extracted algorithm resides in the inefficient manipulation of permutation operations as well as inefficient data structure for the representation of function arguments, incrementing in this way the running time required for the analysis of A and C operators over problems randomly generated as explained. The small effect caused by the addition of C function symbols, for both implementations, is explained by the fact that the inputs with high cost attributed to the C checking are artificial and with low probability of occurrence in the random input generator. Substantial additional running time is required if AC-symbols are included (Figure 12, row 2). Notice that, the time-axis scale on the right is 750 times bigger than on the left. Indeed, in the extracted algorithm, one moves from milliseconds to seconds. This is explained because of the required exhaustive application of the linear running time implementations for the operators $\|\_\|_f$, $\_(\_)_f$ and $\_[\star\_]_f$ (see Figure 4) used to deal with AC terms. This happens since these operators were implemented straightforwardly over tuples that are in fact built as combinations of nominal pairs. In addition, the approach adopted in the improved implementation is more efficient regarding recursive computation of equality checking for arguments of AC operators. The advantages of this approach can be observed in Figure 12, row 2, where the maximum execution time was less than 3 milliseconds for inputs of size around five thousand and less than one hundredth of a second for inputs of size around ten thousand. Accentuation of the curves for inputs of size greater than 8300, for both implementations can be explained by memory saturation; larger

terms could be treated by improving the data structures used for representing arguments of nominal AC operators.

5.4. Upper bounds

Several techniques from [18], originally implemented to deal polynomially with nominal α-equivalence as well as with nominal matching, should be adopted in order to obtain efficient algorithms. Among these techniques, it is necessary to use adequate data structures, such as trees for nominal terms and random access structures for maintaining and answering in constant time queries about the images of permutations and their inverses, as well as for updating compositions of swappings and permutations (and their inverses). The log-linear algorithm defined in [18] to check α-equivalence relies on the use of "lazy permutations": permutations, their inverses and supports are "suspended" over nominal terms and updated eagerly whenever swappings have to be applied, but they are only pushed down one level in the tree structure of the terms when a transformation rule is applied, and they are applied to terms only when necessary.

Remark 1. *To illustrate why such an approach is used, consider lines 10 to 14 in Algorithm 1, related with the application of the rule* $(\approx_\alpha [\mathbf{ab}])$*. Special care has to be taken with* $(a\ b)\cdot t'$ *(line 12, rule* $(\approx_\alpha [\mathbf{ab}])$*), since it is not a term in our syntax, the permutation has to be propagated in* t' *and this introduces an additional linear factor on the complexity of checking* α*-equivalence. However, adopting the above-mentioned approach, where the syntax is extended with "suspended" permutations over terms, which are propagated in a "lazy" way, this linear factor is avoided. Also, notice that there is a secondary check for freshness constraints in* $a \# t'$*. This requires an algorithm for validating freshness constraints based on simplification rules for freshness (Fig. 2 bottom up) which is linear in* $\langle \nabla, a \# t' \rangle$*. To avoid repeated computations (for instance, the check for* $a \# t'$ *may appear several times in the computation) one could append valid freshness constraints in* ∇*, that is, line 12 becomes* $\texttt{Check}(\nabla \cup \{a \# t'\}, \{s' \approx (a\,b)t'\} \cup P')$*.*

Theorem 1 (Running time bounds)**.** *Let* n *be the size of a problem* $\langle \nabla, P \rangle$*, given as* $|\langle \nabla, P \rangle| := |\nabla| + |P|$*, where* $|\nabla|$ *is the number of atoms and variables occurring in* ∇ *and* $|P|$ *the sum of the size of terms in equations in* P*. The validity of* $\langle \nabla, P \rangle$ *modulo A, C and AC can be checked in time*

i) $O(n \log n)$*, if the problem includes neither C nor AC-function symbols;*
ii) $O(n^2 \log n)$*, if the problem does not contain AC function symbols; and*
iii) $O(n^3 \log n)$*, otherwise.*

Proof. (sketch)

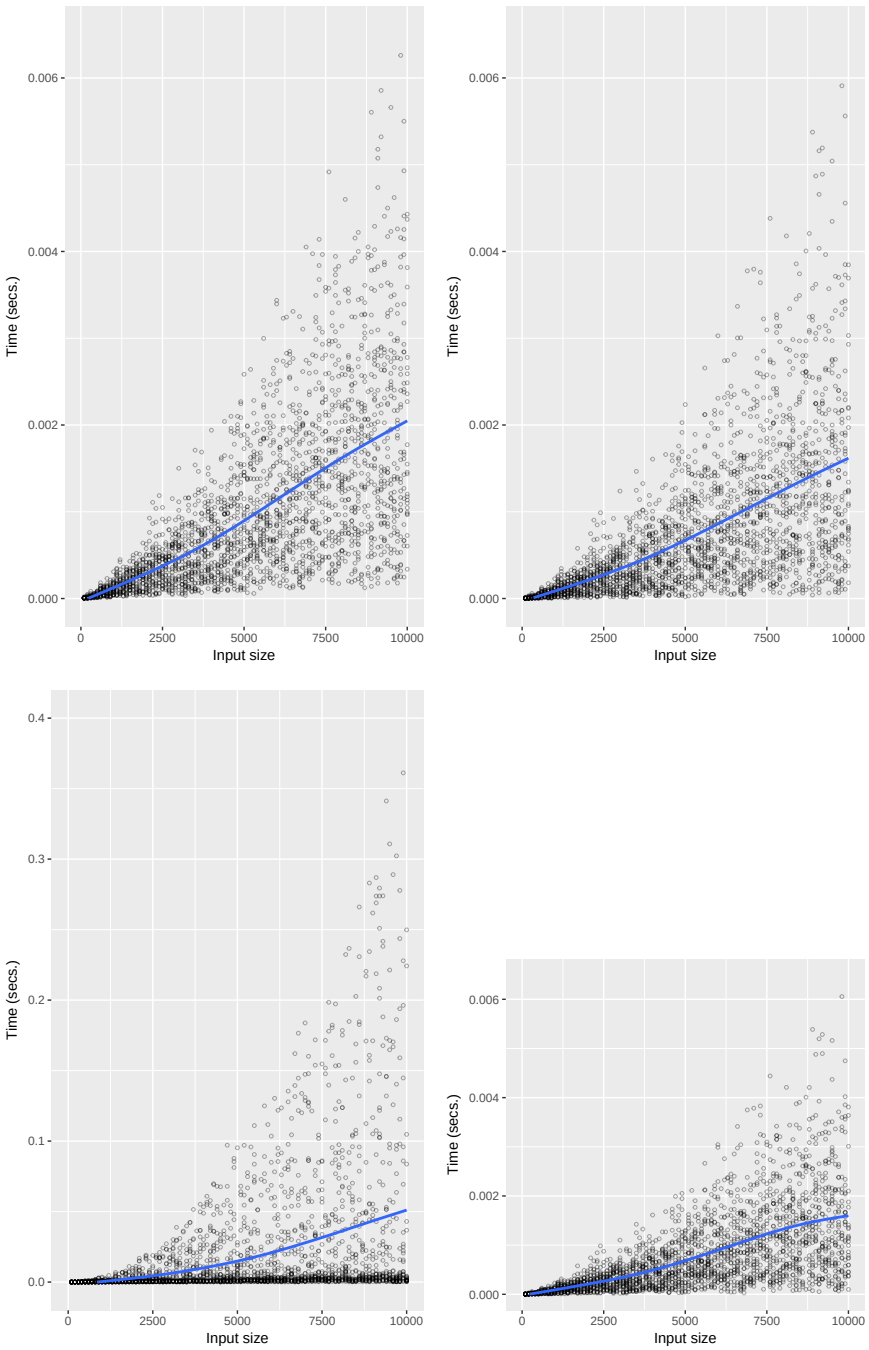

Figure 11: Tests with only α and α-A operators

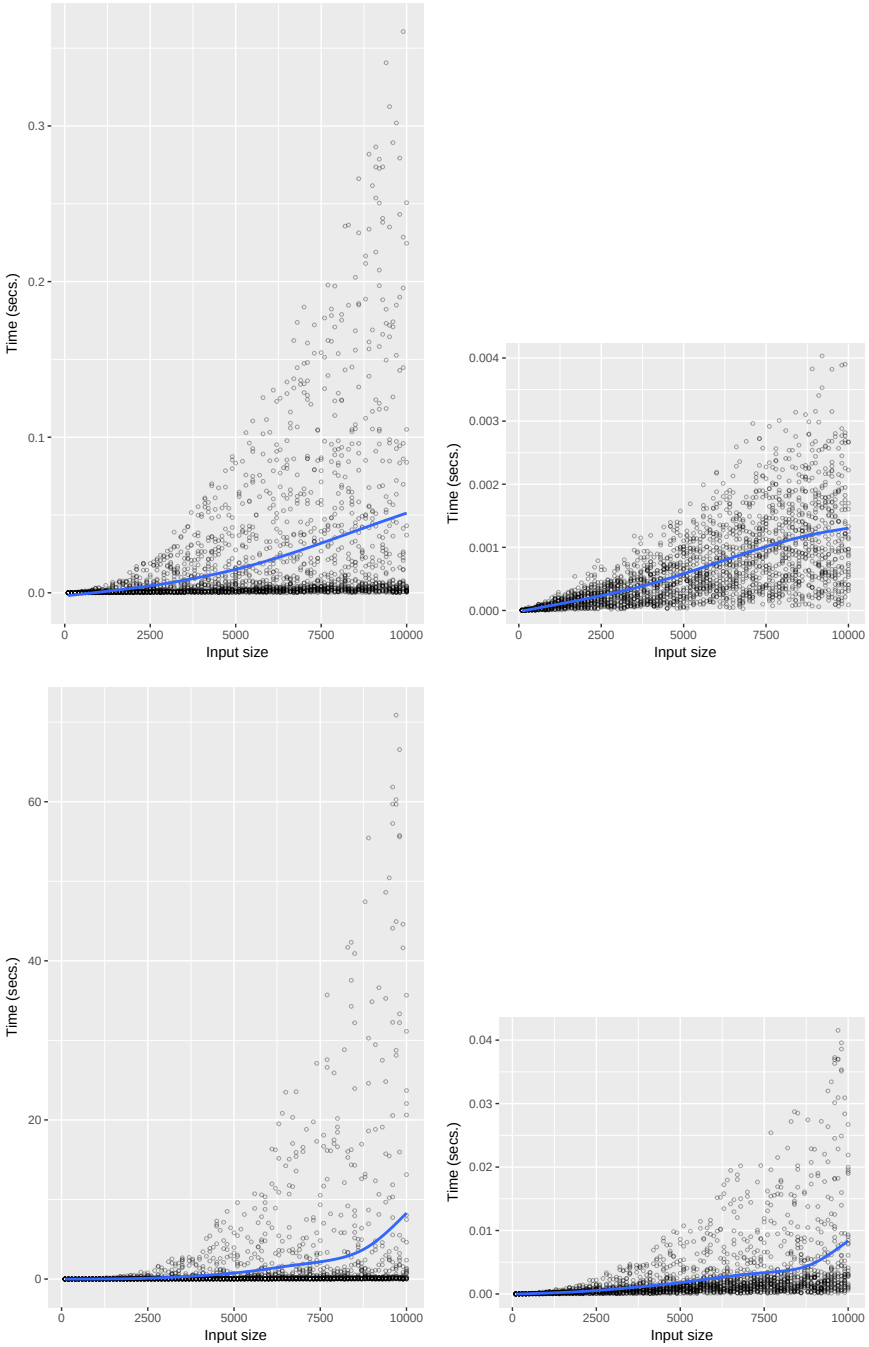

Figure 12: Tests with α-A-C and α-A-C-AC operators

To obtain these bounds we assume first, the use of suspended permutations over terms and of lazy propagation of permutations (see Remark 1); second, that terms in the problems are pre-computed providing a flat representation of the arguments of A-function symbols. For the latter, all maximal subterms that are headed by A-function symbols should be linearly pre-computed to provide their arguments. This can be done for instance using sequences or arrays of terms in which arguments of A-functions are *flattened* and might be accessed randomly (in constant time).

i) Consider a problem of the form $\langle \nabla, \{s \approx t\}\rangle$ where s and t contain neither C- nor AC-function symbols. Since A-function symbols are assumed to be flattened, the problem can be log-linearly solved through a simple adaptation of the solution for α-equivalence checking given in [18]. For the A case, the problem can be directly decomposed, according to the number n_s of flattened arguments, into a new problem with n_s new disjoint equational sub-problems, that is, a problem of the form $\langle \nabla, P \cup \{f^A_k s' \approx f^A_k t'\}\rangle$ becomes directly a problem of the form $\langle \nabla, P \cup \{s'_{(1)_{f^A_k}} \approx t'_{(1)_{f^A_k}}, \ldots, s'_{(n_s)_{f^A_k}} \approx t'_{(n_s)_{f^A_k}}\}\rangle$.

ii) Let $\langle \nabla, \{s \approx t\}\rangle$ be a problem without AC-function symbols. A regular worst case happens when the problem has k nested C-function symbols. Assume that $n = m\,2^k$, where $m \ll n$. In this case, if we consider terms with the same commutative symbol at the root, and with arguments of the same size, an upper bound for the running time is given by the recurrence relation:

$$T(n) = 4\,T(n/2) + O(n\,\log n)\ \textit{where}\ T(m) = O(m \log m).$$

In the first recurrence equation, the first summand has a factor 4 because it is necessary to check four sub problems of half of the original size, and the second summand provides a bound, according to the previous item, if the term does not have C-operators. Notice that both summands are included since the objective is to give an upper bound. The initial condition of the recurrence relation also assumes that sub-problems of size m have no occurrences of C-function symbols. Thus one has,

$$\begin{aligned}
T(n) &= 4\,T(n/2) + O(n \log n)\\
&= 4^k\,T(m) + O(n)\textstyle\sum_{i=0}^{k-1} 2^i(\log n - i \log 2)\\
&= (\tfrac{n}{m})^2 O(m \log m) + O(n \log n)\textstyle\sum_{i=0}^{k-1} 2^i - O(n)\log 2 \sum_{i=0}^{k-1} 2^i i\\
&= O(n^2 \tfrac{\log m}{m}) + O(n \log n)(2^k - 1) - O(n)\log 2(2^k(k-2)+2)\\
&= O(n^2) + O(n^2 \log n)\\
&= O(n^2 \log n)
\end{aligned}$$

Factors related with m can be omitted since we assume that $m \ll n$.

iii) First, notice that terms headed by C-function symbols can be considered as a particular case of AC symbols whose tuples (arguments) have always exactly two elements. Thus, the complexity analysis for C- and AC-function symbols could be unified. Let $\langle \nabla, \{s \approx t\} \rangle$ be a problem that contains AC-function symbols. Assuming the flat representation of all maximal subterms of s and t that are headed with A and AC-function symbols is pre-computed, the relevant part of the analysis is related with the verification of α-equivalence between subterms s' and t' of s and t headed by an AC-function symbol, say f_k^{AC}. This involves checking whether the tuple of n_s arguments in s' contains arguments that are related by α-equivalence modulo AC to arguments of the tuple of arguments in t'. These arguments are not necessarily in the same positions in the tuples of arguments of s' and t'. In the worst case scenario, for each argument of the tuple of arguments of f_k^{AC} in s', say $s'_{(i)_{f_k^{AC}}}$, one has to go over the whole tuple of arguments of f_k^{AC} in t', checking $\langle \nabla, \{s'_{(i)_{f_k^{AC}}} \approx t'_{(j)_{f_k^{AC}}}\} \rangle$, for $i, j \leq ||s'||_{f_k^{AC}}$. In case this is true, α-equivalence eliminating these two arguments of the tuples should be checked. By item i), one already knows that the procedure without C and AC symbols is log-linear. The problem essentially boils down to the problem of searching a perfect matching in the bipartite graph that consists of vertices V labelled by the n_s arguments of the *lhs*'s and *rhs*'s and edges, E, between vertices labelled with terms that match, as proved in [19] for solving AC-matching in the usual first-order syntax. This problem is known to have solutions of complexity $O(|V| \times |E|)$, that is the same as $O(|V|^3)$ since in the worst case one has $O(|V|^2)$ edges [23]. One concludes that searching for a perfect matching is bounded cubically on the size of the problem, since the number of arguments, $||s'||_{f_k^{AC}}$, is linearly bounded in the size of the problem. But notice that for the case of just AC-equivalence, applying this method requires only complexity $O(|V|^2)$ since after having checked two terms to be equivalent, the corresponding edge can be fixed and checking for other equivalences for these terms is unnecessary. Thus, an upper bound for the whole problem is $O(n^3 \log n)$.

□

Remark 2. *Regarding item i), notice that even without function symbols (just atoms, abstractions and tuples) the α-equality check is log-linear for non-ground terms.*

6. Related work

Equational problems have been extensively explored since the early development of modern abstract algebra (see, e.g., the E-unification survey by Baader et al [24]).

Specifically for AC equality checking, AC matching and AC unification problems, refined techniques have been applied. For instance, AC-equality check and linear AC-matching problems can be reduced to searching a perfect matching in a bipartite graph [19], whereas AC unification problems can be reduced to solving a system of Diophantine equations [25].

Formalisations of equational reasoning modulo A, C and AC are available: Nipkow [26] proposed a set of rules that implement rewriting tactics in Isabelle/HOL to reason modulo A/C/AC. This set was used to build equational matching and unification algorithms, but aspects of performance and termination of these algorithms were not explored. Contejean [27] developed a sound and complete A/C/AC-matching algorithm that was defined as a set of rewriting rules that decompose equations until *solved* normal forms are reached. This algorithm was formalised in Coq and implemented in C*i*ME, but efficiency and complexity analysis were not provided. Additionally, Braibrant and Pous [28] designed a plugin for Coq to use the tactic `rewrite` modulo A/AC. The development of tactics `aac_rewrite`, which uses matching modulo A/C, and `aac_reflexivity` which uses equality checking modulo A/AC, was based on the `Morphisms` library and an auxiliary OCaml program with implementations of the equality checking and matching algorithms. Proofs of soundness of the algorithms were provided, but, again, neither complexity analysis nor performance tests were performed.

Checking validity of α-equivalence constraints has been studied in [18], where an algorithm to test α-equivalence of nominal terms (both ground or non-ground), derived from a *core algorithm* to solve matching problems modulo α, was provided. The matching algorithm is linear in the size of the problem for the ground case (i.e., when matching a term s against a ground term t) and therefore α-equivalence is also linear in this case. If both terms are non-ground, then α-equivalence is log-linear in the size of the problem, whereas matching is log-linear if the pattern is linear and quadratic otherwise.

Beyond the nominal unification formalisation of Urban et al. [6, 7], there are also other formal nominal developments in Isabelle/HOL, Coq, HOL4, PVS and Agda. For example, Aydemir, Bohannon and Weirich developed nominal reasoning techniques in Coq [15]. The authors investigated principles of induction and recursion modulo α-equivalence following the nominal approach. However, the specification diverges from the nominal approach since the core syntax of terms uses indices to represent bound object level variables. Urban [16] proposed a framework in Isabelle/HOL that also allows reasoning modulo nominal α-equivalence, defining principles of induction and recursion modulo α-equivalence in which a higher-order argument is used to deal with abstracted object-level variables. Finally, Copello et al. [29] presented a nominal approach, based essentially on nominal swapping and freshness, used to deal in a concrete manner with α-conversion in the λ-calculus.

This was used in order to obtain a formalisation in Agda of principles of structural induction and recursion modulo α, without using indexes or higher-order expressions to represent bound object-level variables.

Regarding nominal unification, Kumar and Norrish [4] presented a nominal unification algorithm that uses *descent recursion* and *triangular substitutions*, with underlying formalisations in HOL4 of the correctness and termination of the proposed algorithm. Ayala-Rincón, Fernández and Rocha-Oliveira [5] formalised in PVS the soundness of the nominal $\approx_\alpha$-equivalence relation, and soundness and completeness of a nominal unification algorithm. Finally, Ayala-Rincón et al. [30] applied this development in order to formalise soundness and completeness of a nominal unification algorithm modulo C.

A few distinguishing elements of nominal formalisation developments are listed below. In the following, except in our current Coq formalisation, only pure reasoning over the nominal syntax is explored.

- The current Coq formalisation, as the Isabelle/HOL formalisation of nominal unification in [6, 7], inductively specifies equational procedures as sets of inference rules. These sets are given through inductive predicates, which allow the proof assistants to build inductive proof schemes on the predicates in a straightforward manner. In this approach, this is convenient since it allows the construction of a sole inductive definition with specific rules for all desired equational theories handled by parameters that determine which rules are active. This allows a modular treatment of equational check in signatures with operators with different equational properties such as A, C, AC, and eventually others and their combinations.

- In contrast to the inductive approaches used in our Coq development and the Isabelle/HOL referenced approach, the HOL4 and PVS formalisations of nominal unification in [4] and [5], respectively, use a recursive style to specify unification algorithms. In these developments inductive proofs are guided by smart termination measures provided as part of the specification. In the PVS development a first-order functional algorithm *à la* Robinson was specified and verified sound and correct, which has as parameter only pairs of nominal terms (i.e., equations), but no freshness constraints. Avoiding freshness constraints as parameters is one of the distinguishing features of this PVS formalisation, that is possible due to the formalisation of properties on the independence of freshness contexts regarding substitutions in solutions. The HOL4 formalisation specifies triangular substitutions that are sets of singleton bindings for different variables used to present unification in an accumulator-passing style, in which in the execution of each recursive call a substitution is taken as input returning an extension on success. Both of these

recursive specifications allow extraction of recursive unification functions, but they do not allow the modularity of the inductive approaches in which new inference rules can be added and fragments of the previous correctness proofs (concerning the analysis of cases related with previous existing inference rules) can be reused.

Regarding extensions of nominal equational reasoning, nominal narrowing was introduced by Ayala-Rincón, Fernández and Nantes-Sobrinho in [31]. This work adapts Hullot's seminal work on narrowing, originally developed from the first-order rewriting perspective, to the nominal approach in such a manner that nominal equational unification problems are solvable by narrowing whenever the equational theories can be presented as a class of convergent closed rewriting systems.

Another extension of nominal unification was proposed in Schmidt-Schauss et al. [32]. This development proposed an algorithm to solve nominal unification problems with *recursive let* operators. In this algorithm, the solutions of a unification problem are expressed in terms of nominal fixed point equations. Obtaining solutions for such equations is a recurrent problem; indeed, in [30] it was shown that nominal C-unification problems are reduced to solving finite families of fixed point equations. This work also proved that nominal C-unification is infinitary, differing from syntactic C-unification that is well-known to be finitary. Afterwards, in [33], Ayala-Rincón et al. proposed a sound and complete combinatorial procedure to generate the set of solutions of nominal fixed point problems. Recently, Ayala-Rincón, Fernández and Nantes-Sobrinho [34] proposed a representation of solutions of nominal C-unification problems based on *fixed-point constraints*. The authors showed that the standard representation of solutions, composed by a pair of a freshness context and a substitution, can be translated conservatively to a pair of fixed-point constraints and a substitution. Following this approach, nominal C-unification problems are finitary, as in the syntactic case.

7. Conclusion and Future Work

The soundness of nominal α-equivalence and its extension to the equational theories A, C, AC and their combinations were formalised in Coq.

Checking soundness of these relations required checking that they are specified in such a manner that they are indeed equivalence relations. In particular, the property of transitivity of $\approx_\alpha$ was formalised in a direct manner without using an auxiliary weak intermediate relation as done in [4], [6] and [17]. The proof of transitivity follows the approach introduced in [5] for formalising nominal unification in PVS. Here, the proof is based on elementary lemmas about permutations, freshness and α-equivalence; such lemmas are well-known in the context of nominal unification.

In [6], the same auxiliary lemmas to demonstrate transitivity were proved, including some extra lemmas to deal with this weak-equivalence. The current formalisation of transitivity of $\approx_\alpha$ is simpler in the sense that it only uses the essential notions and results. Indeed, adopting the direct approach in [5] resulted in a more compact formalisation with several improvements, among them a formalisation of symmetry of $\approx_\alpha$ that is independent from transitivity, diverging from the approach that uses weak equivalence, where it is obtained as a consequence of transitivity.

The grammar of nominal terms was specified in such a way that in addition to A, C and AC rules one can easily add other inference rules to express properties such as *idempotency* (I), *neutral* (U) and *inverse* elements (**G**roup theory), and their combinations A, AC, AI, ACI, ACU, ACUI, etc.

Enriching nominal α-equality with equational theories formally, will provide an effective framework for dealing not only with nominal α-equivalence, but also with other related fundamental relations such as nominal *matching*, *unification* and *narrowing* in concrete applications. Examples of such applications can be found in several contexts, such as the one of integrity of cryptographic protocols [31, 35, 36]. A further interesting analysis would be the classification of the related problems of nominal α-matching and unification modulo theories, regarding their complexities.We have started this investigation with the case of α-unification modulo C in [30, 33, 34] and we are currently implementing efficient versions of the α-equivalence decision algorithm to use within a nominal matching algorithm modulo A, C and AC theories.

References

[1] H. Barendregt, The Lambda Calculus: Its Syntax and Semantics, revised ed., vol. 103 of Studies in Logic and the Foundations of Mathematics, North-Holland, 1984. doi:10.2307/2274112.

[2] A. M. Pitts, Nominal Logic, a First Order Theory of Names and Binding, Information and Computation 186 (2) (2003) 165–193. doi:10.1016/S0890-5401(03)00138-X.

[3] C. F. Calvès, M. Fernández, *Implementing Nominal Unification*, ENTCS 176 (1) (2007) 25–37. doi:10.1016/j.entcs.2006.09.027.

[4] R. Kumar, M. Norrish, *(Nominal) Unification by Recursive Descent with Triangular Substitutions*, in: Proc. of the 1st Int. Conf. of Interactive Theorem Proving (ITP), Vol. 6172 of LNCS, Springer, 2010, pp. 51–66. doi:10.1007/978-3-642-14052-5_6.

[5] M. Ayala-Rincón, M. Fernández, A. C. Rocha-oliveira, *Completeness in PVS of a Nominal Unification Algorithm*, ENTCS 323 (2016) 57–74. doi:10.1016/j.entcs.2016.06.005.

[6] C. Urban, *Nominal Unification Revisited*, in: Proc. of the 24th Int. Work. on Unification (UNIF), Vol. 42 of EPTCS, 2010, pp. 1–11. doi:10.4204/EPTCS.42.1.

[7] C. Urban, A. M. Pitts, M. J. Gabbay, *Nominal Unification*, Theoretical Computer Science 323 (1-3) (2004) 473–497. doi:10.1016/j.tcs.2004.06.016.

[8] M. Fernández, M. J. Gabbay, *Nominal Rewriting*, Information and Computation 205 (6) (2007) 917–965. doi:10.1016/j.ic.2006.12.002.

[9] M. Fernández, M. J. Gabbay, *Closed nominal rewriting and efficiently computable nominal algebra equality*, in: Proc. of the 5th Int. Work. on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP), Vol. 34 of EPTCS, 2010, pp. 37–51. doi:10.4204/EPTCS.34.5.

[10] M. Fernández, M. J. Gabbay, I. Mackie, *Nominal Rewriting Systems*, in: Proc. of the 6th Int. Conf. on Principles and Practice of Declarative Programming (PPDP), ACM Press, 2004, pp. 108–119. doi:10.1145/1013963.1013978.

[11] J. Cheney, *αProlog Users Guide - Version 0.3 DRAFT*, available at http://homepages.inf.ed.ac.uk/jcheney/programs/aprolog/guide.pdf (2003).

[12] W. E. Byrd, D. P. Friedman, αKanren: A Fresh Name in Nominal Logic Programming, In Proc. of the Workshop on Scheme and Functional Programming (2007) 79–90.

[13] M. R. Shinwell, The Fresh Approach: functional programming with names and binders, Tech. rep., University of Cambridge, available at https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-618.pdf (2005).

[14] M. R. Shinwell, A. M. Pitts, M. J. Gabbay, FreshML: Programming with binders made simple, in: Proc. of the Int. Conference on Functional Programming (ICFP), ICFP'03, ACM Press, 2003, pp. 263–274. doi:10.1145/944705.944729.

[15] B. Aydemir, A. Bohannon, S. Weirich, Nominal Reasoning Techniques in Coq, ENTCS 174 (5) (2007) 69–77. doi:dx.doi.org/10.1016/j.entcs.2007.01.028.

[16] C. Urban, Nominal Techniques in Isabelle/HOL, J. of Autom. Reasoning 40 (4) (2008) 327–356. doi:10.1007/s10817-008-9097-2.

[17] M. Ayala-Rincón, W. de Carvalho Segundo, M. Fernández, D. Nantes-Sobrinho, A formalisation of nominal α-equivalence with A and AC function symbols, ENTCS 332 (2017) 21–38. doi:10.1016/j.entcs.2017.04.003.

[18] C. F. Calvès, M. Fernández, Matching and Alpha-Equivalence Check for Nominal Terms, J. of Computer and System Sciences 76 (5) (2010) 283–301. doi:http://dx.doi.org/10.1016/j.jcss.2009.10.003.

[19] D. Benanav, D. Kapur, P. Narendran, Complexity of Matching Problems, J. of Sym. Computation 3 (1/2) (1987) 203–216. doi:10.1016/S0747-7171(87)80027-5.

[20] M. Sozeau, Equations: A Dependent Pattern-Matching Compiler, in: Proc. of the 1st Int. Conf. of Interactive Theorem Proving (ITP), Vol. 6172 of LNCS, Springer, 2010, pp. 419–434. doi:10.1007/978-3-642-14052-5_29.

[21] M. Sozeau, Subset Coercions in Coq, in: Proc. of the Int. Work. on Types for Proofs and Programs (TYPES), Vol. 4502 of LNCS, Springer, 2006, pp. 237–252. doi:10.1007/978-3-540-74464-1_16.

[22] D. Larchey-Wendling, J.-F. Monin, Simulating Induction-Recursion for Partial Algorithms, accepted to TYPES. Available at https://members.loria.fr/DLarchey/files/papers/TYPES_2018_paper_19.pdf (2018).

[23] T. H. Cormen, C. E. Leiserson, R. Rivest, C. Stein., Introduction to Algorithms, The MIT Press, 2009.

[24] F. Baader, W. Snyder, P. Narendran, M. Schmidt-Schauß, K. U. Schulz, Unification Theory, in: Handbook of Automated Reasoning (in 2 volumes), MIT Press and North Holand, 2001, pp. 445–454.

[25] F. Fages, Associative-Commutative Unification, J. of Sym. Computation 3 (1987) 257–275. doi:10.1016/S0747-7171(87)80004-4.

[26] T. Nipkow, *Equational Reasoning in Isabelle*, Science of Computer Programming 12 (2) (1989) 123–149. doi:10.1016/0167-6423(89)90038-5.

[27] E. Contejean, *A Certified AC Matching Algorithm*, in: Proc. of the 15th Int. Conf. on Rewriting Techniques and Applications (RTA), Vol. 3091 of LNCS, Springer, 2004, pp. 70–84. doi:10.1007/978-3-540-25979-4_5.

[28] T. Braibant, D. Pous, *Tactics for Reasoning Modulo AC in Coq*, in: In Proc. of the 1st. Int. Conf. on Certified Programs and Proofs (CPP), Vol. 7086 of LNCS, Springer, 2011, pp. 167–182. doi:10.1007/978-3-642-25379-9_14.

[29] E. Copello, E. Tasistro, N. Szasz, A. Bove, M. Fernández, *Principles of Alpha-Induction and Recursion for the Lambda Calculus in Constructive Type Theory*, ENTCS 323 (2016) 109–124. doi:10.1016/j.entcs.2016.06.008.

[30] M. Ayala-Rincón, W. Carvalho-Segundo, M. Fernández, D. Nantes-Sobrinho, *Nominal C-Unification*, in: Proc. of the 27th Int. Symp. Logic-Based Program Synthesis and Transformation (LOPSTR), Vol. 10855 of LNCS, Springer, 2017, pp. 235–251. doi:10.1007/978-3-319-94460-9_14.

[31] M. Ayala-Rincón, M. Fernández, D. Nantes-Sobrinho, *Nominal Narrowing*, in: Proc. of the 1st Int. Conf. on Formal Structures for Computation and Deduction (FSCD), Vol. 52 of LIPIcs, SDLZI, 2016, pp. 11:1–11:17. doi:10.4230/LIPIcs.FSCD.2016.11.

[32] T. Kutsia, J. Levy, M. Schmidt-Schauß, M. Villaret, *Nominal Unification of Higher Order Expressions with Recursive Let*, in: Proc. of the 26th Int. Symp. on Logic-Based Program Synthesis and Transformation (LOPSTR), Vol. 10184 of LNCS, Springer, 2016, pp. 328–344. doi:10.1007/978-3-319-63139-4_19.

[33] M. Ayala-Rincón, W. Carvalho-Segundo, M. Fernández, D. Nantes-Sobrinho, *On Solving Nominal Fixpoint Equations*, in: Proc. of the 11th Int. Symp. on Frontiers of Combining Systems (FroCoS), Vol. 10483 of LNCS, Springer, 2017, pp. 209–226. doi:10.1007/978-3-319-66167-4_12.

[34] M. Ayala-Rincón, M. Fernández, D. Nantes-Sobrinho, *Fixed-Point Constraints for Nominal Equational Unification*, in: Proc. of the 3rd Int. Conf. on Formal Structures for Computation and Deduction (FSCD), Vol. 108 of LIPIcs, SDLZI, 2018, pp. 7:1–7:16. doi:10.4230/LIPIcs.FSCD.2018.7.

[35] V. Cortier, S. Delaune, P. Lafourcade, *A survey of algebraic properties used in cryptographic protocols*, J. of Computer Security 14 (1) (2006) 1–43.

[36] S. Escobar, C. Meadows, J. Meseguer, Maude-NPA: Cryptographic Protocol Analysis Modulo Equational Properties, Foundations of Security Analysis and Design 5705 (2007) 1–50. doi:10.1007/978-3-642-03829-7_1.